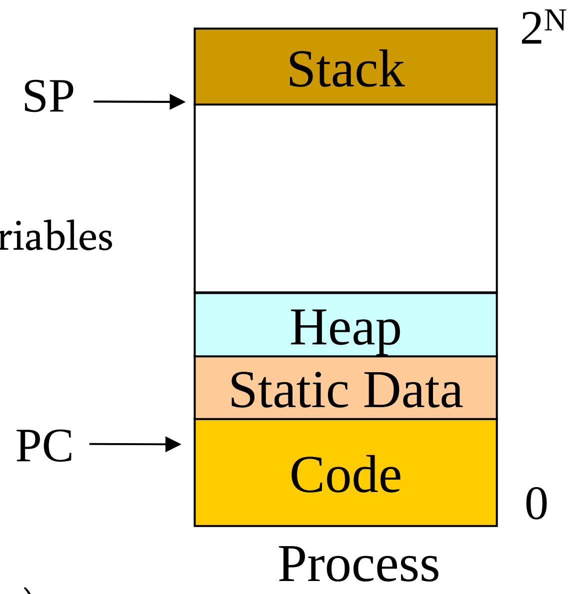
Introduction to Concurrent Programming Using Processes and Pthreads

Professor: Panagiota Fatourou
TA: Eleftherios Kosmas
CSD - March 2012

Process

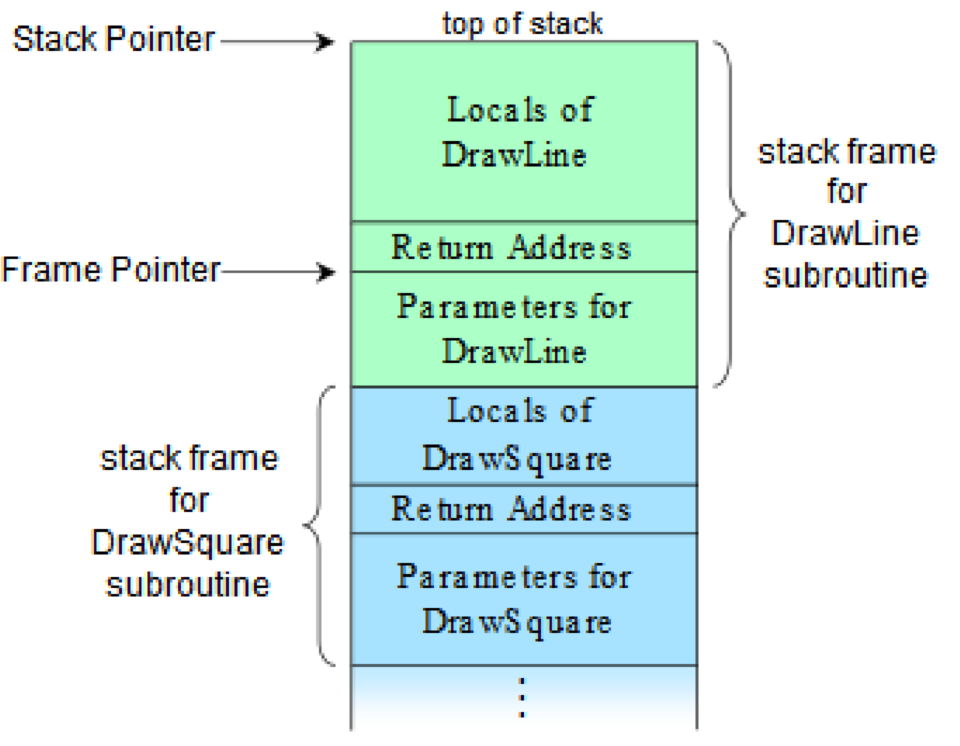
- A process is an **instance** of a program that is being executed
- It consists of, or it owns: (PCB - Process Control Block)
  - an executable machine code
  - memory (some private address space)
    - input and output data
    - call stack: keeps track of active subroutines
      - function parameters, return addresses, and local variables
    - heap: dynamically allocated memory
    - static data: global variables
    - file descriptors, socket descriptors, etc.
    - processor state (or context),
      - e.g., content of CPU registers (Program Counter - PC, Stack Pointer - SP, numeric)



- Allows program to act as if it owns the machine
- Multitasking using **timesharing**
  - context switch, scheduling

Process - Call Stack

- It is composed of stack frames or activation records
- Each stack frame
  - corresponds to an active subroutine
  - is machine dependent



Multitasking

- Parent processes create children processes by calling `fork()`
- **`pid_t fork()`**
  - spawns a new child process
    - with separate address space
    - it has an exact copy of all the memory segments of the parent process
    - it has a new pid
  - returns
    - 0 to child
    - the pid of the newly created child to parent
- **`pid_t wait(int *status)`**
  - suspends execution of the calling process until one of its children terminates
  - status indicates reason for termination

Multitasking

- **`pid_t waitpid (pid_t pid, int *status, int options)`**
  - pid=-1: any child process
  - pid>0: specific child process
  - pid=0: any child process with some process group id
  - options can be used to wait or to check and procced

- **`int exit (int *status)`**
  - executed by a child process when it wants to terminate
  - makes status available to parent

Multitasking - Example

```
#include <stdio.h>   /* printf, stderr, fprintf */
#include <sys/types.h> /* pid_t */
#include <unistd.h>  /* _exit, fork */
#include <stdlib.h>  /* exit */
#include <errno.h>   /* errno */

int main(void)
{
  pid_t pid;
  int i;

  for (i=0; i<10; i++) {
    /* Output from both the child and the parent process will be written to the standard output, as they both run at the same time. */
    pid = fork();
    if (pid == -1) {
      /* Error: When fork() returns -1, an error happened (for example, number of processes reached the limit). */
      fprintf (stderr, "can't fork, error %d\n", errno);
      exit (EXIT_FAILURE);
    }
    else if (pid = 0) break; /* When fork() returns 0, we are in the child process. */
  }

  if (pid == 0) { /* When fork() returns 0, we are in the child process. */
    ...
    exit(0);
  }
  else { ...        /* When fork() returns a positive number, we are in the parent process */
```

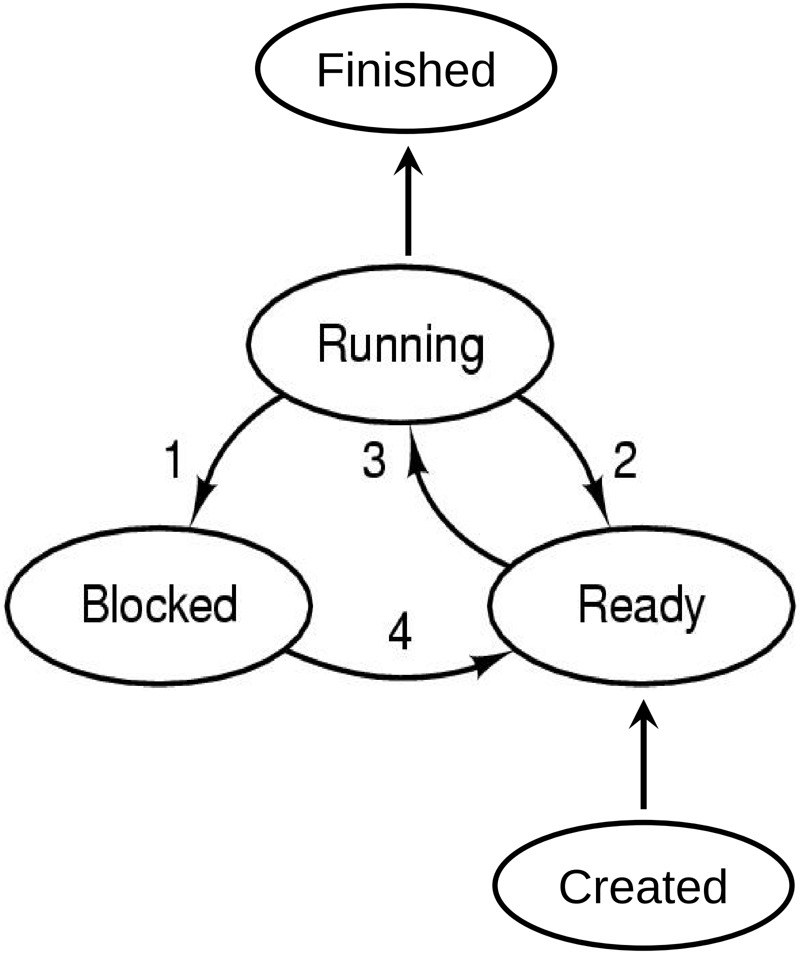
Multitasking - Example

```
#include <stdio.h>   /* printf, stderr, fprintf */
#include <sys/types.h> /* pid_t */
#include <unistd.h>  /* _exit, fork */
#include <stdlib.h>  /* exit */
#include <errno.h>   /* errno */

int main(void)
{
  pid_t pid;
  for (i=0; i<10; i++) {
   pid = fork();
   …
   else if (pid = 0) break; /* When fork() returns 0, we are in the child process. */
  }

  if (pid == 0) {   /* Child process: When fork() returns 0, we are in the child process. */
    ...
    exit(0);
  }
  else {             /* When fork() returns a positive number, we are in the parent process */
    for(i=0; i<10; i++) {
      pid_t child = wait(0);
      printf (“Child with pid [%d] terminated\n", child);
    }
  }
  return 0;
}
```

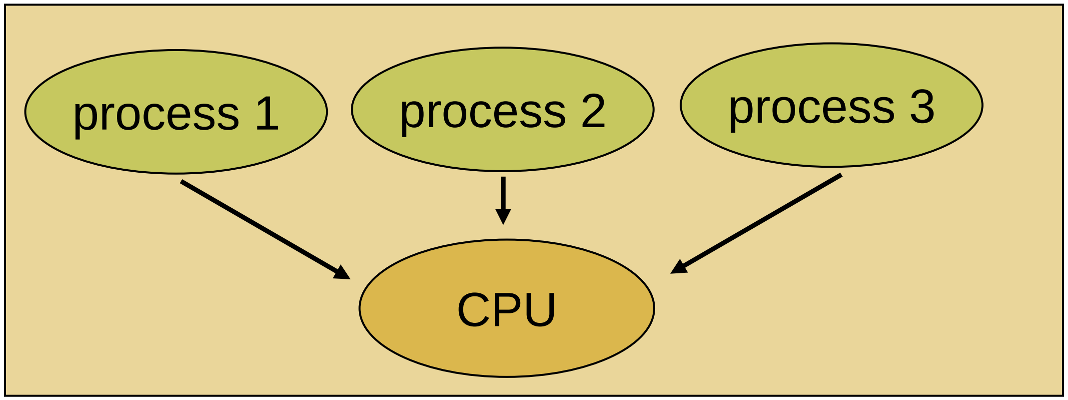
Multitasking - Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

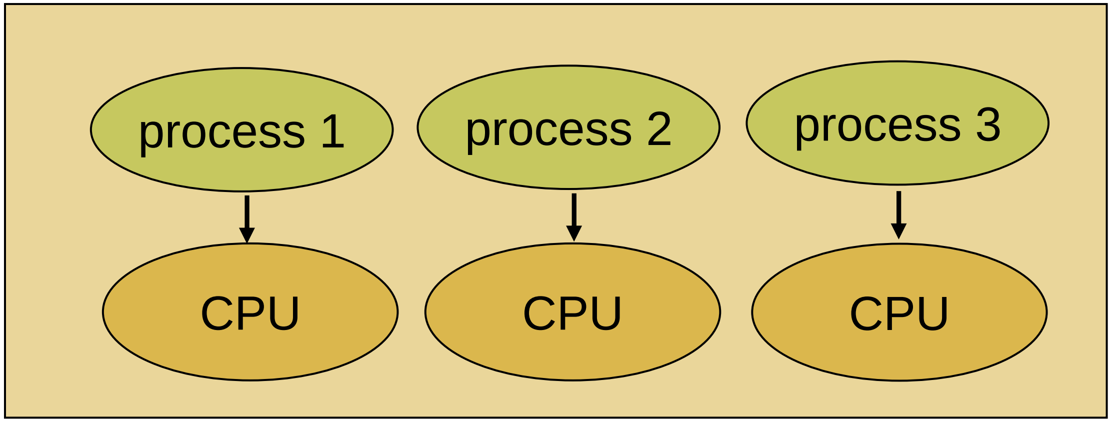
Multiprocessing

- When applying multitasking to a multicore (multiprocessor) machine we get **multiprocessing**



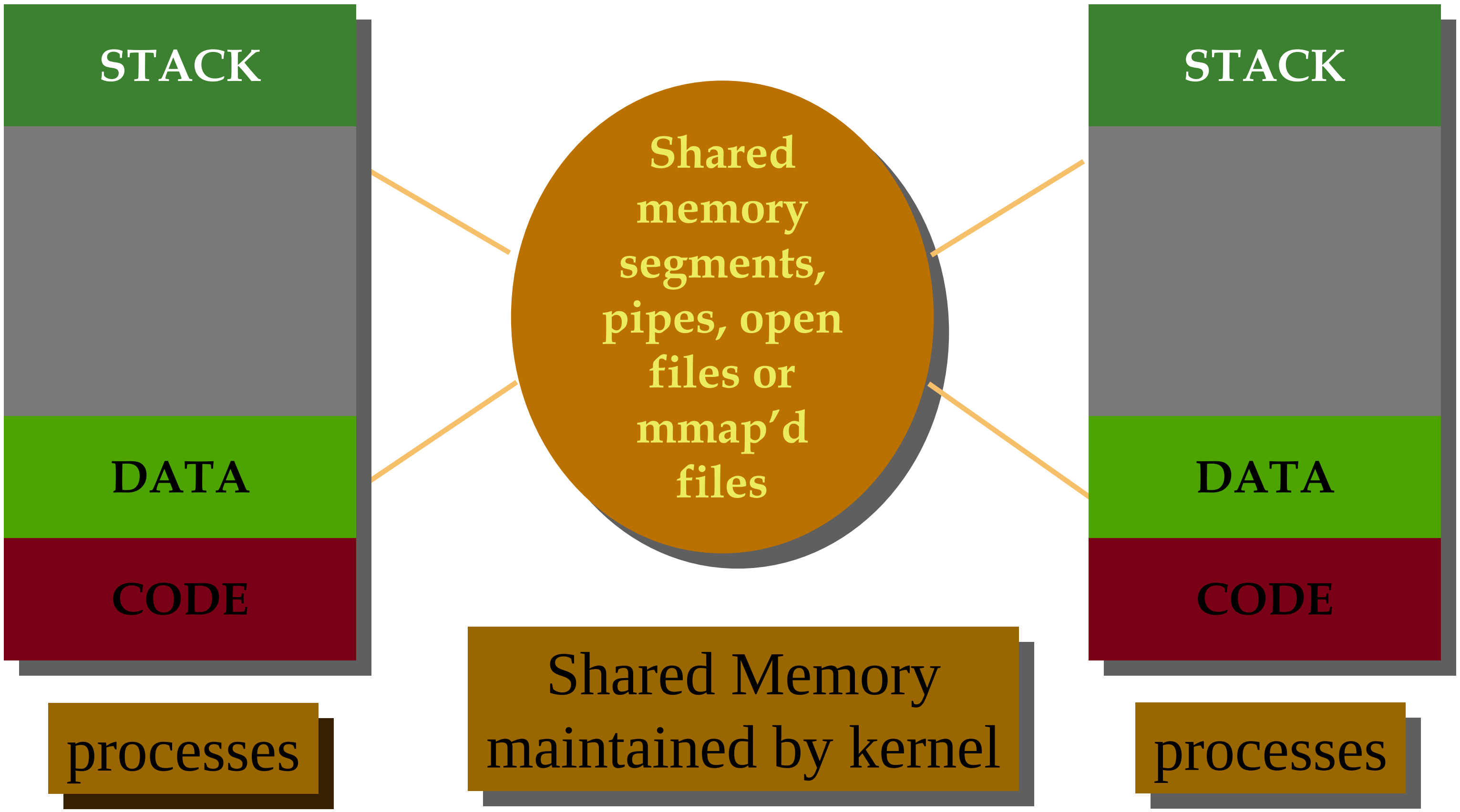
Multitasking

Multiprocessing



Inter Process Communication (IPC)

- processes do not share anything implicitly
- explicit actions are required to achieve IPC



Shared Memory IPC

- One process will create a memory portion which other processes (if permitted) can access

- `shmget()`: is used to create a shared memory segment
  - a shared memory segment is described by a control structure (in `<sys/shm.h>`) with a unique ID that points to an area of physical memory
- `shmctl()`:
  - used by the original owner of a shared memory segment can assign (or revoke) ownership to another user
  - used by processes with permission to perform various control functions
- `shmat()`: attach a shared segment to a process address space
  - Once attached, the process can read or write to the segment
- `shmdt()`: detach

Shared Memory IPC

- `int shmget(key_t key, size_t size, int shmflg)`:
  - **returns** the identifier of shared memory segment associated with the value of the argument key
  - it is also used to get the identifier of an **existing** shared segment
  - size: the size in bytes of the requested shared memory
  - shmflg: specifies the initial access permissions and creation control flags

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
...
key_t key; int shmflg; int shmid; int size;
...
key = ...; size = ...; shmflg = ...;
if ((shmid = shmget (key, size, shmflg)) == -1) {
   perror("shmget: shmget failed"); exit(1);
}
else {
   fprintf(stderr, "shmget: shmget returned %d\n", shmid); exit(0);
}
...
```

Shared Memory IPC

- `int shmctl(int shmid, int cmd, struct shmid_ds *buf)`:
  - is used to alter the permissions and other characteristics of a shared memory segment
  - cmd: SHM_LOCK, SHM_UNLOCK, IPC_STAT, IPC_SET, IPC_RMID
  - buf: shared memory data structure to hold results

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
...
int cmd; int shmid; struct shmid_ds shmid_ds;
...
shmid = ...; cmd = ...;
if ((rtrn = shmctl(shmid, cmd, shmid_ds)) == -1) {
    perror("shmctl: shmctl failed");
    exit(1);
   }
...
```

Shared Memory IPC

- `void *shmat(int shmid, const void *shmaddr, int shmflg):`
  - **returns** a pointer, shmaddr, to the head of the shared segment associated with a valid shmid
  - shmflag: flags used on attach
- `int shmdt(const void *shmaddr):`
  - detaches the shared memory segment located at the address indicated by shmaddr

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

static struct state {
          int shmid; char *shmaddr; int shmflg;
} ap[MAXnap]; /* State of current attached segments. */
int nap; /* Number of currently attached segments. */
...
char *addr; /* address work variable */
register int i; /* work area */
register struct state *p; /* ptr to current state entry */
...
p = &ap[nap++];
p->shmid = ...; p->shmaddr = ...; p->shmflg = ...
```

Shared Memory IPC

- `void *shmat(int shmid, const void *shmaddr, int shmflg):`
- `int shmdt(const void *shmaddr):`

```
p->shmaddr = shmat(p->shmid, p->shmaddr, p->shmflg);
if(p->shmaddr == (char *)-1) {
    perror("shmop: shmat failed");
    nap--;
}
else fprintf(stderr, "shmop: shmat returned %#8.8x\n", p->shmaddr);
...
i = shmdt(addr);
if(i == -1) {
    perror("shmop: shmdt failed");
}
else {
    fprintf(stderr, "shmop: shmdt returned %d\n", i);
    for (p = ap, i = nap; i--; p++)
        if (p->shmaddr == addr) *p = ap[--nap];
}
...
```

Shared Memory IPC - Example - Server

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    char c, *shm, *s; int shmid; key_t key;

    key = 1234;
    shmid = shmget(key, SHMSZ, IPC_CREAT | 0666);
    shm = shmat(shmid, NULL, 0);

    s = shm;
    for (c = 'a'; c <= 'z'; c++)
        *s++ = c;
    *s = NULL;

    while (*shm != '*')
        sleep(1);

    shmctl(shmid, IPC_RMID, 0)
    exit(0);
}
```

Shared Memory IPC - Example - Client

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>

#define SHMSZ     27

main()
{
    int shmid; key_t key; char *shm, *s;

    key = 1234;
    shmid = shmget(key, SHMSZ, 0666);
    shm = shmat(shmid, NULL, 0);

    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');

    *shm = '*';

    exit(0);
}
```

Threads

- A process is the **heaviest** unit of kernel scheduling
  - **creating** a new process is costly
    - data structures needed to be allocated and initialized
  - expensive **context switch**
  - **communication** among processes is costly, since it goes through the OS
    - IPC
    - overhead of system calls and copying data

- A process consists of:

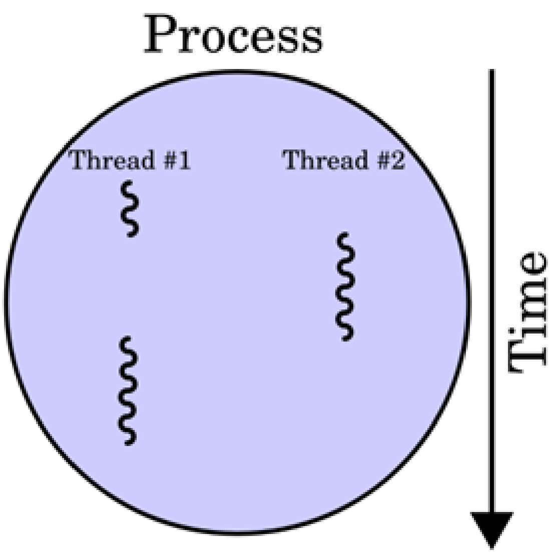
  i. a collection of resources
    - the code & address space, open files, etc.

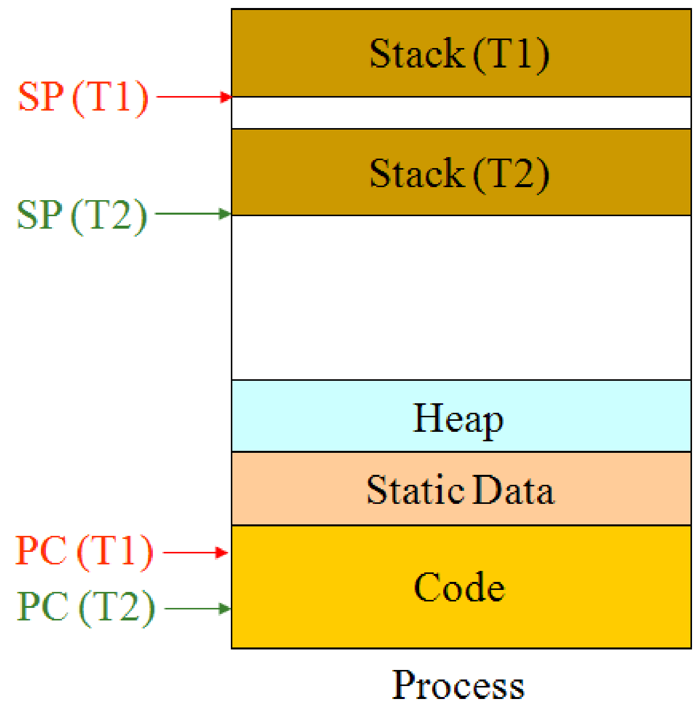
  ii. a **thread** of execution
    - the current state that operates on these resources

☞ The idea is to let multiple threads share a common address space

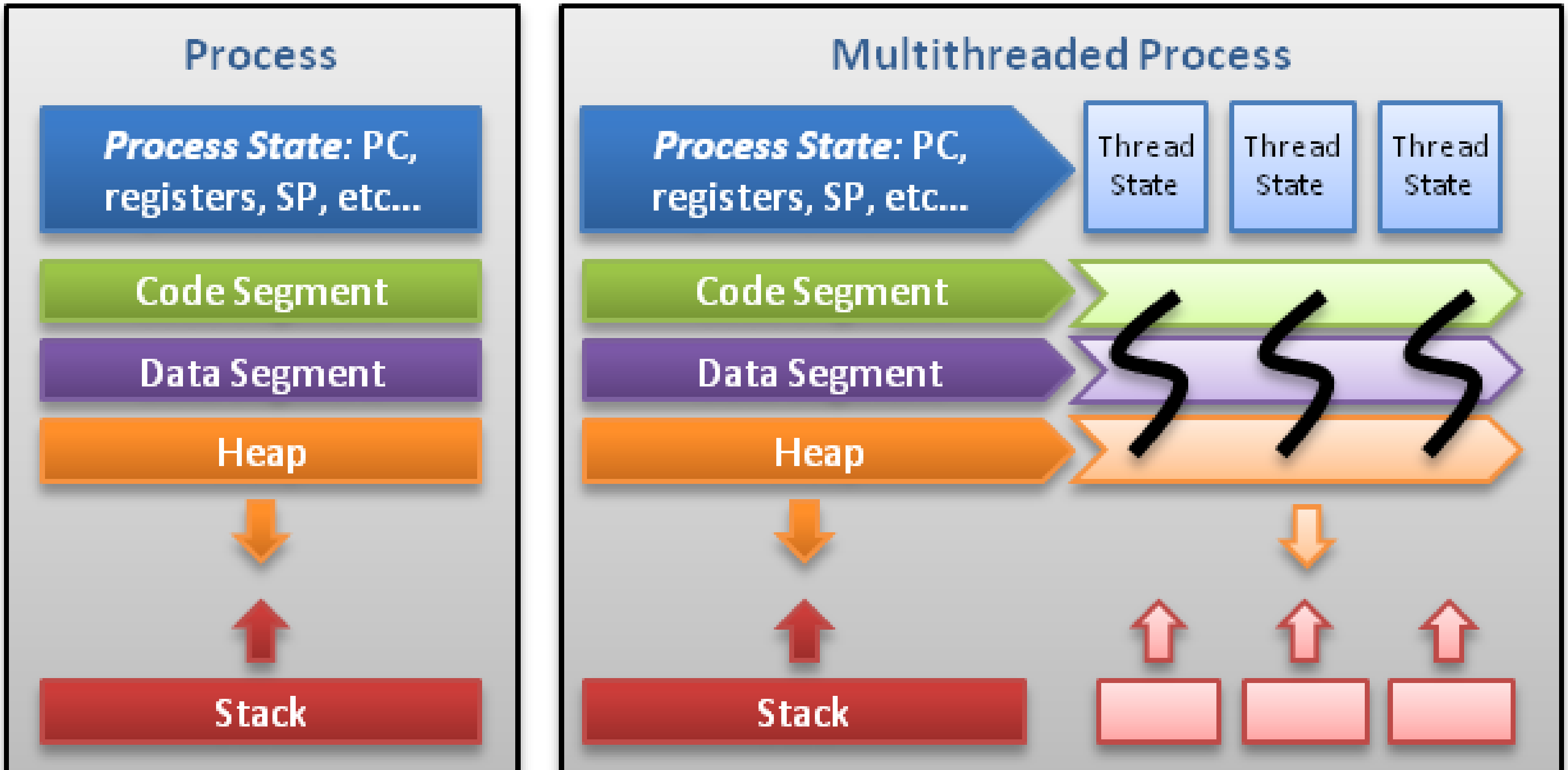
Threads

- Threads share the same memory (global variables, heap, file descriptors, etc.)
- Threads own a stack (including thread- local storage) and a copy of the registers (including PC and SP)
- Threads are executed in parallel
  - using time slices, in a single core machine
  - or really in parallel, in a multicore machine





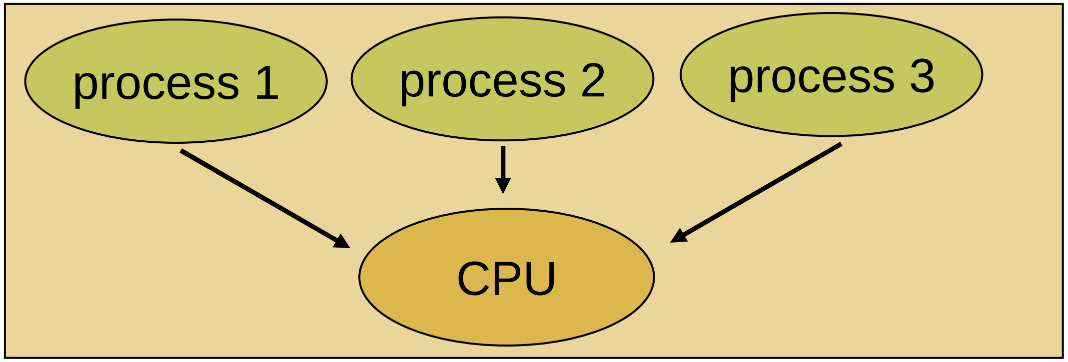
Single vs. Multi threaded



Threads contain only necessary information, such as a stack (for local variables, function arguments, return values), a copy of the registers, program counter and any thread-specific data to allow them to be scheduled individually. Other data is shared within the process between all threads.

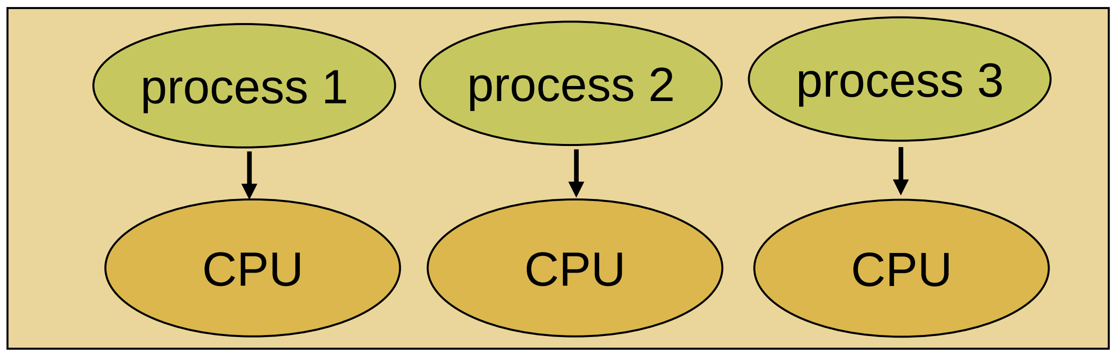
Multithreading

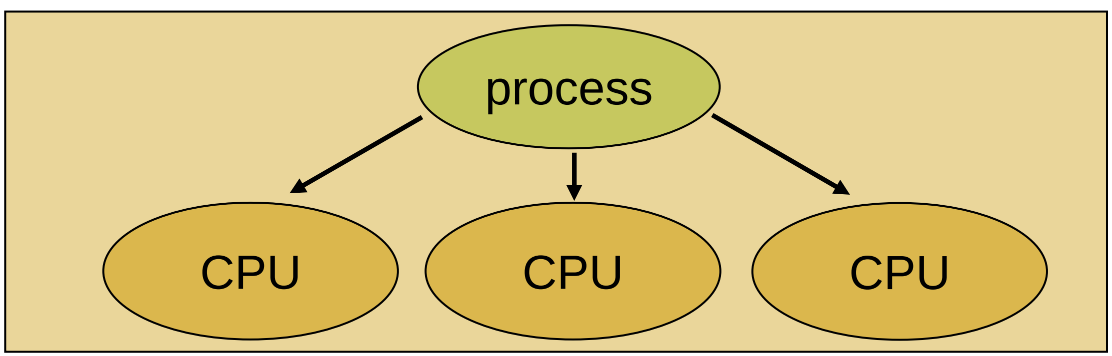
- A way for program to split itself into multiple running tasks



Multitasking

Multiprocessing





Multithreading

Threads vs. Processes

- Threads
  - ✓ easier to create and destroy
  - ✓ inter-thread communication is cheaper
    - can use process memory and may not need (for user-level threads) to context switch
  - ✓ provide faster context switch
  - ✗ not secure: a thread can write the memory used by another thread
- Processes
  - ✓ secure: one process cannot corrupt another process
  - ✗ inter-process communication is expensive: need to context switch

Threads

- A kernel thread is the lightest unit of kernel scheduling
- Each process contains at least one kernel thread

- The kernel
  - may (on may not) assign one thread to each logical core, resulting to different models:
    - 1:1, **Kernel-level threading**
    - N:1, **User-level threading**
    - M:N, Hybrid threading
  - can swap out threads that get blocked
  - ✘ kernel threads take much longer than user threads to be swapped

Threads

- **User threads** are managed and scheduled in userspace
  - may run on top of several kernel threads to benefit from multi-processors
  - ✓ fast to create and manage
  - ✗ can not take full advantage of multithreading
    - ✗ they get blocked when all of their associated kernel threads are blocked, even if there are some user threads that are ready to run

POSIX Threads or Pthreads API

- ***Thread management:*** The first class of functions work directly on threads - creating, terminating, joining, etc.
- ***Mutexes:*** provide for creating, destroying, locking and unlocking mutexes.
- ***Condition variables:*** include functions to create, destroy, wait and signal based upon specified variable values.
- ***Barriers***

Pthreads - Thread Management

- `int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`:
  - thread: the actual thread object that contains pthread id
  - attr: attributes to apply to this thread
  - start_routine: the function this thread executes
  - arg: arguments to pass to thread function above

- `void pthread_exit(void *value_ptr)`:
  - terminates the thread and provides value_ptr available to any `pthread_join()` call

- `int pthread_join(pthread_t thread, void **value_ptr)`:
  - suspends the calling thread to wait for successful termination of thread
  - value_ptr: data passed from the terminating thread's call to `pthread_exit()`

Pthreads - Thread Management - Example

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NUM_THREADS 2

                     /*create thread argument struct for thr_func() */
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

void *thr_func(void *arg) {   /* thread function */
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

  pthread_exit(NULL);
}
```

Pthreads - Thread Management - Example

```
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS];
  int i, rc;
                                   /* create a thread_data_t argument array */
  thread_data_t thr_data[NUM_THREADS];
  for (i = 0; i < NUM_THREADS; ++i) {     /* create threads */
    thr_data[i].tid = i;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }

  for (i = 0; i < NUM_THREADS; ++i)  /* block until all threads complete */
    pthread_join(thr[i], NULL);

  return EXIT_SUCCESS;
}
```

Pthreads - Thread Management

- **`int pthread_create(pthread_t *thread, pthread_attr_t *attr, void *(*start_routine)(void *), void *arg)`**:
  - attr: attributes to apply to this thread

☞ Attributes can be **specified** using the following functions:

- `int pthread_attr_init(pthread_attr_t *attr)`

- `int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate)`
- `int pthread_attr_setguardsize(pthread_attr_t *attr, size_t guardsize)`
- `int pthread_attr_setinheritsched(pthread_attr_t *attr, int inheritsched)`
- `int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param)`
- `int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)`
- `int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope)`
- `int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr)`
- `int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize)`

☞ Attributes can be **retrieved** via corresponding get functions

Pthreads - Mutexes

- `int pthread_mutex_init (pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr):`
  - initializes **mutex**
  - attributes for the mutex can be given through **mutexattr**
    - use NULL, to specify default attributes

- `int pthread_mutex_lock(pthread_mutex_t *mutex)`
  - blocks until mutex lock is acquired

- `int pthread_mutex_trylock(pthread_mutex_t *mutex)`
  - non-blocking, may return without acquiring the mutex lock

- `int pthread_mutex_unlock(pthread_mutex_t *mutex)`

Pthreads - Mutexes - Example

```
...
typedef struct _thread_data_t {
  int tid;
  double stuff;
} thread_data_t;

double shared_x;               /* shared data between threads */
pthread_mutex_t lock_x;        /* shared data between threads */

void *thr_func(void *arg) {    /* thread function */
  thread_data_t *data = (thread_data_t *)arg;

  printf("hello from thr_func, thread id: %d\n", data->tid);

                      /* get mutex before modifying and printing shared_x */
  pthread_mutex_lock(&lock_x);
  shared_x += data->stuff;
  printf("x = %f\n", shared_x);
  pthread_mutex_unlock(&lock_x);

  pthread_exit(NULL);
}
```

Pthreads - Mutexes - Example

```
int main(int argc, char **argv) {
  pthread_t thr[NUM_THREADS]; int i, rc;
  thread_data_t thr_data[NUM_THREADS];

  shared_x = 0; /* initialize shared data */
                /* initialize pthread mutex protecting "shared_x" */
  pthread_mutex_init(&lock_x, NULL);

  for (i = 0; i < NUM_THREADS; ++i) {        /* create threads */
    thr_data[i].tid = i;
    thr_data[i].stuff = (i + 1) * NUM_THREADS;
    if ((rc = pthread_create(&thr[i], NULL, thr_func, &thr_data[i]))) {
      fprintf(stderr, "error: pthread_create, rc: %d\n", rc);
      return EXIT_FAILURE;
    }
  }

  for (i = 0; i < NUM_THREADS; ++i)  /* block until all threads complete */
    pthread_join(thr[i], NULL);

  return EXIT_SUCCESS;
}
```

Pthreads - Condition Variables

- **`int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr)`**:
  - initialized condition variable cond
  - attributes for cond can be given through cond_attr
    - use NULL, to specify default attributes

- **`int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)`**
  - puts the current thread to sleep, waiting on cond for mutex to be released

- **`int pthread_cond_signal(pthread_cond_t *cond)`**
  - signals one thread out of the possibly many sleeping threads waiting on cond to wakeup

- **`int pthread_cond_broadcast(pthread_cond_t *cond)`**
  - signals all threads waiting on cond to wakeup

Pthreads - Condition Variables - Example

```
void *thr_func1(void *arg) {
  pthread_mutex_lock(&count_lock); /*thread code blocks here until MAX_COUNT is reached*/
  while (count < MAX_COUNT)
     pthread_cond_wait(&count_cond, &count_lock);
  pthread_mutex_unlock(&count_lock);
  ...
  pthread_exit(NULL);
}

/*some other thread code that signals a waiting thread that MAX_COUNT has been reached*/
void *thr_func2(void *arg) {
  pthread_mutex_lock(&count_lock);

  /* some code here that does interesting stuff and modifies count */

  if (count == MAX_COUNT) {
    pthread_mutex_unlock(&count_lock);
    pthread_cond_signal(&count_cond);
  }
  else pthread_mutex_unlock(&count_lock);

  pthread_exit(NULL);
}
```

Pthreads - Barrier

- **`int pthread_barrier_init(pthread_barrier_t *barrier, pthread_barrierattr_t *barrier_attr, unsigned int count)`**
  - initialized barrier
  - attributes for barrier can be given through barrier_attr
    - use NULL, to specify default attributes
  - count: defines the number threads that must join the barrier for the barrier to reach completion and unblock all threads waiting at the barrier

- **`int pthread_barrier_wait(pthread_barrier_t *barrier)`**

Pthreads - Additional Useful Functions

- **`pthread_kill()`**, can be used to deliver signals to specific threads.
- **`pthread_self()`**, returns a handle on the calling thread.
- **`pthread_equal()`**, compares for equality between two pthread ids
- **`pthread_once()`**, can be used to ensure that an initializing function within a thread is only run once.

Pthreads - How to compile

- C : **`gcc -lpthread multithreaded_app.c`**
- C++ : **`g++ -lpthread multithreaded_app.cxx`**
- Other platforms and compilers may require **`-pthread`** flag instead