1 Limited Direct Execution

- Idea: Just run the program you want to run on the CPU, but first make sure to set up the hardware so as to limit what process can do without OS assistance
- baby proofs the CPU by

  1. Setting up trap handlers
  2. Starts an interrupt timer
  3. Run processes in a restricted mode

  Example

  Baby proofing a room:

  - Locking cabinets containing dangerous stuff and covering electrical sockets.
  - When room is readied, let your baby roam free in knowledge that all the dangerous aspect of the room is restricted

2 Trap Handlers

- Is instruction that tells the hardware what to run when certain exceptions occur

  Example

  What code to run when

  1. Hard disk interrupt occurs
  2. Keyboard interrupt occrs
  3. Program makes a system call?

3 Timer Interrupt

- Is a hardware mechanism that ensures the user program does not run forever
- Is emitted at regular intervals by a timer chip [6]

4 Response Time

- **Formula** $T_{response} = T_{firstrun} - T_{arrival}$
- measures the interactive performance between users and the system

5 Turnaround Time

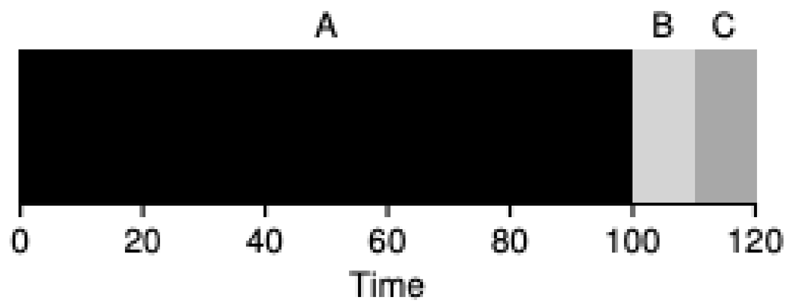
- **Formula** $T_{turnaround} = T_{completion} - T_{arrival}$
- measures the amount of time taken to complete a process

6 Starvation

- Is the problem that occurs when high priority processes keep executing and low priority processes get blocked for indefinite time [1]

7 Convoy Effect

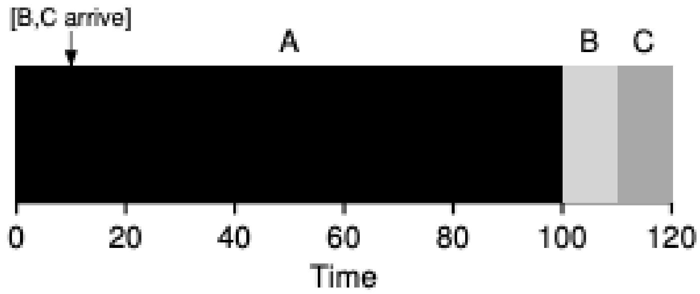
- Is the problem where number of relatively-short potential consumers of a resource get queued behind a heavy weight consumer



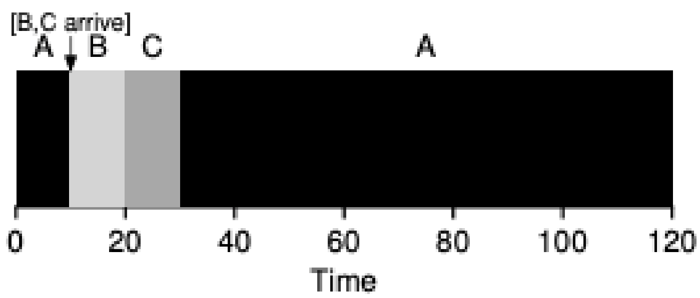
8 Scheduling policies

- Are algorithms for allocating CPU resources to concurrent tasks deployed on (i.e., allocated to) a processor (i.e., computing resource) or a shared pool of processors [5]
- Are sometimes called **Discipline**
- Covers the following algorithms in textbook
  - **First In First Out**
    * Is the most basic scheduling algorithm
    * Is vulnerable to **convoy effect**
    * No **starvation** as long as every process eventually completes
  - **Shortest Job First**
    * Improves average **turnaround time** given processes of uneven length
    * Is a general scheduling principle useful in situation where turnaround time per process matters

    * Is vulnerable to **convoy effect**



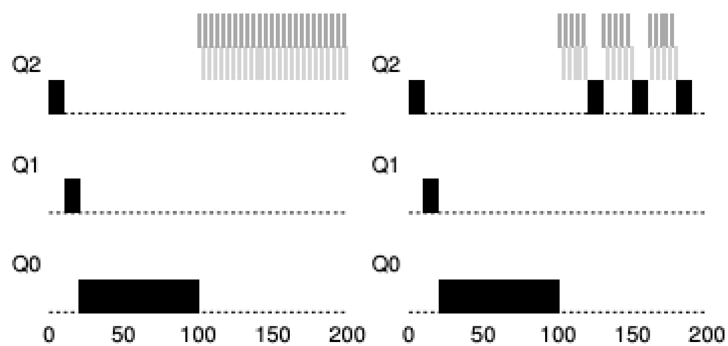
    * Is vulnerable to **starvation**
        · When only short-term jobs come in while a long term job is in queue
  - **Shortest Time-to-completion First**
    * Addresses **convoy effect** in **Shortest Job First**
    * Determines which of the remaining+new jobs has least time left, and schedule accordingly at <u>any time</u>



    * Is vulnerable to **starvation**
        · When only short-term jobs come in while a long term job is in queue
  - **Round Robin**
    * Has good **response time** but terrible **turnaround time**
    * Runs job for a **time slice** or **quantum**
    * Each job gets equal share of CPU time
    * Is clock-driven [6]
    * Is starvation-free [7]
    * <u>Must</u> have the length of a time slice (**quantum**) as multiple of timer-interrupt period

```
void release(boolean *lock) {
        *lock = false;
}
```

- **Multi-level Feedback Queue**
  - Is the most well known approaches to shceduling
  - Optimizes **turnaround time**, and minimizes **response time**
  - Observees the execution of a job and priortizes accordingly without prior knowledge
  - Rules
    - **Rule 1:** If Priority(A) > Priority(B), A runs (B doesn't)
    - **Rule 2:** If Priority(A) = Priority(B). A & B run in round-robin fashion using the time slice (quantum length) of the given queue
    - **Rule 3:** When a job enters the system, it is placed at the highest priority(the top most queue)
    - **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (it moves down on queue)
    - **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.



9 User Mode

- Is restricted
- Executing code has no ability to *directly* access hardware or reference memory [1]
- Crashes are always recoverable [1]
- Is where most of the code on our computer / applications are executed [3]

10 Kernel Mode

- Is previleged (non-restricted)
- Executing code has complete and unrestricted access to the underlying hardware [3]

- Is generally reserved for the lowest-level, most trusted functions of the operating system [1]
- Is fatal to crash; it will halt the entire PC (i.e the blue screen of death) [3]

11 Interrupt

- i is a signals are sent by hardware (keyboardm mouse, etc.), or software (page fault, protection violation, system call)
- Tells the CPU to stop its current activities and execute the appropriate part of the operating system (**Interrupt Handler**). [2]
- Has three different types [2]

  1) **Hardware Interupts**
     - Are generated by hardware devices to signal that they need some attention from the OS.
     - May be due to receiving some data

       Examples

       * Keystrokes on the keyboard
       * Receiving data on the ethernet card

     - May be due to completing a task which the operating system previous requested

       Examples

       Transfering data between the hard drive and memory

  2) **Software Interupts**
     - Are generated by programs when a system call is requested

  3) **Traps**
     - Are generated by the CPU itself
     - Indicate that some error or condition occured for which assistance from the operating system is needed

12 Content Switch

- Is switching from running a user level process to the OS kernel and often to other user processes before the current process is resumed
- Happens during a timer interrupt or system call
- Saves the following states for a process during a context switch
  - Stack Pointer
  - Program Counter
  - User Registers
  - Kernel State
- May hinder performance

13 System Call

- Is the programmatic way in which a computer program requests a previleged service from the kernel of the operating system
- i.e. Reading from disk
- Is strictly a subset of software interrupts
- Steps
  1) Setup **trap tables** on boot
  2) Execute system call
  3) Save *Program Counter*, *CPU registers*, *kernal stack* (so process can resume after **return-from-trap** or **context switch**)
  4) Switch from **user mode** to **kernel mode**
  5) Perform previleged operations
  6) Finish and execute **return-from-trap** instruction
  7) Return from **kernel mode** to **user mode** and resume user program

Example

- `yield()`
  - Is a system call
  - Causes the calling thread to relinquish the CPU
  - Places the current thread at the end of the run queue
  - Schedules another thread to run

14 Signals

- Provides a way to communicate with the process
- Can cause job to stop, continue, or terminate
- Can be delivered to an application
  - Stops the application from whatever its doing
  - Runs Signal handler (some code in application to handle the signal)
  - When finished, the process resumes previous behavior

15 CPU-bound process

[8]

- CPU Bound processes are ones that are implementing algorithms with a large number of calculations
- Programs such as simulations may be CPU bound for most of the life of the process.
- Users do not typically expect an immediate response from the computer when running CPU bound programs.
- They should be given a lower priority by the scheduler.

16 I/O-bound process

[8]

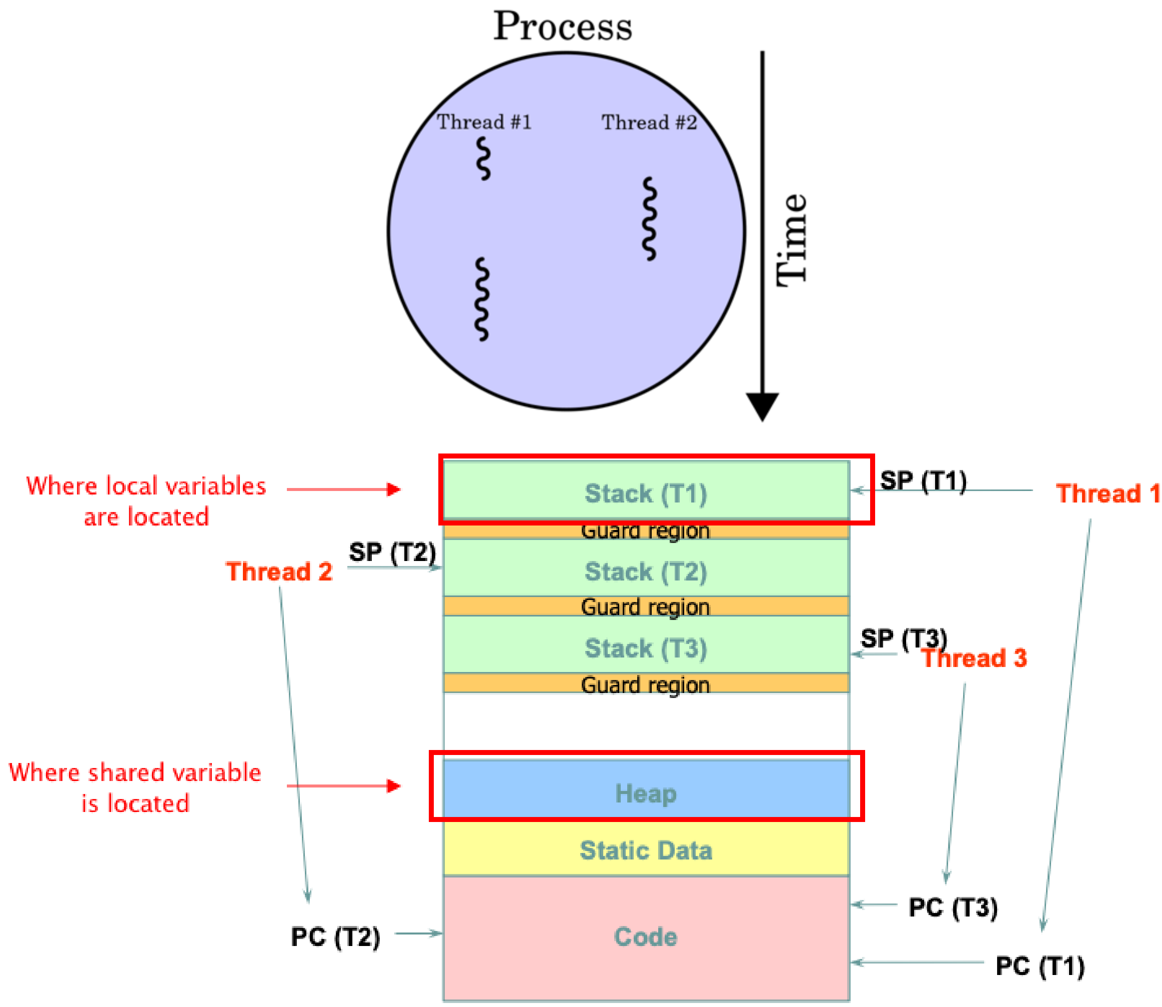
- Processes that are mostly waiting for the completion of input or output (I/O) are I/O Bound.
- Interactive processes, such as office applications are mostly I/O bound the entire life of the process. Some processes may be I/O bound for only a few short periods of time.
- The expected short run time of I/O bound processes means that they will not stay the running the process for very long.
- They should be given high priority by the scheduler.

17 Critical Section

- Is a piece of code that accesses a *shared* resource, usually a variable or data structure

18 Thread

- Is a lightweight process that can be managed independently by a schdeduler [4]
- Improves the application performance using parallelism. (e.g peach)



- A thread is bound to a single process
- A process can have multiple threads
- Has two types
  - **User-level Threads:**
    * Are implemented by users and kernel is not aware of the existence of these threads
    * Are represented by a program counter(PC), stack, registers and a small process control block

    * Are small and much faster than kernel level threads
  - **Kernel-level Threads:**
    * Are handled by the operating system directly
    * Thread management is done by the kernel
    * Are slower than user-level threads

19 Thread API

- `pthread_create`
  - **syntax:**

```
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void * (*start_routine)(void*),
                   void * arg)
```

    * `thread`
      - is a pointer to a structure of type `pthread_t`
    * `attr`
      - is used to specify any attributes this thread might have
      - is initialized with a separate call `pthread_attr_init()`
      - set default by passing `NULL`
    * `(start_routine)`
      - means which function this thread should start running in?
      - setting void pointer `(void *)` as an argument to function `start_routine` allows us to pass in <u>any</u> type of argument
      - setting void pointer `(void *)` as return type allows us to return <u>any</u> type of result
    * `args`
      - is where to pass the arguments for the function pointer ((start_routine))

    <u>Example</u>

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

typedef struct {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Struct is copied here

Argument is initialized here

- `pthread_cond_wait`
  - Puts the calling thread to sleep (a blocked state)
  - Waits for some other thread to signal it

Example

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

Puts calling thread cond to sleep

- `pthread_cond_signal`
  - Is used to unblocks at least one of the threads that are blocked on the specified condition variable cond

Example

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

Wakes a thread that's been put to sleep on cond variable

20 Condition Variable

- is an explicit queue that threads can put themselves on when some state of execution is not as desired (so it can be put to sleep)
- when states are changed, one or more of the waiting threads can be awaken and be allowed to continue (done by **signaling** the condition)
- queue is **FIFO**
- `wait()` call is used to put thread to sleep
- `singal()` call is used to awake thread from sleep
- **Syntax (initialization):**

  `pthread_cont_t c = PTHREAD_COND_INITIALIZER`

  Example

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Initialized here

- **Syntax (Wait):**

  `Pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`

  <u>**Example**</u>

```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Put thread to sleep until done

- **Syntax (Signal):**

  `Pthread_cond_signal(pthread_cond_t *c)`

  Example

```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Awake thread here

21 Spinlock

- Is the simplest lock to build
- Uses a lock variable
  - 0 - (available/unlock/free)
  - 1 - (acquired/locked/held)
- Has two operations
  1. `acquire()`

```
boolean test_and_set(boolean *lock)
{
      boolean old = *lock;
      *lock = True;
      return old;
}
boolean lock;

void acquire(boolean *lock) {
       while(test_and_set(lock));
}
```

2. `release()`

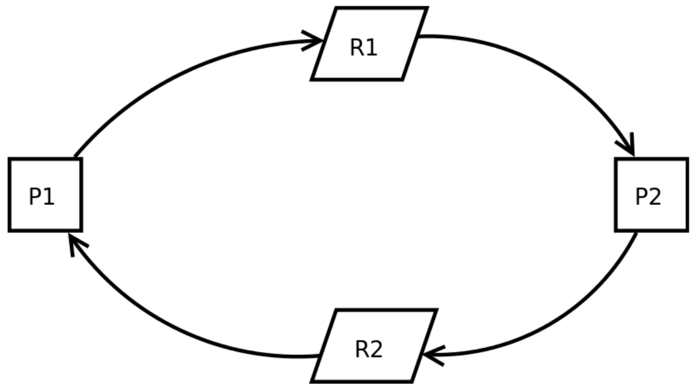
```
void release(boolean *lock) {
       *lock = false;
}
```

- Allows a single thread to enter critical section at a time
- Spins using CPU cycles until the lock becomes available.
- May spin forever

22 Livelock

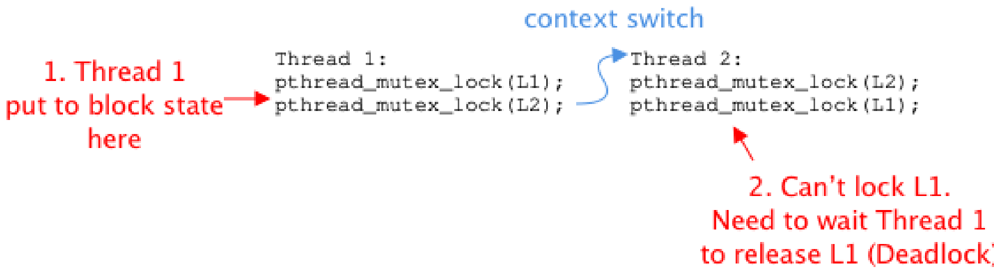
- Two or more threads reapeatedly attempting this code over and over (e.g acquiring lock), but progress is not being made (e.g acquiring lock)
- Solution: Add a random delay before trying again (decrease odd of livelock)

23 Deadlock



- Is a state in which each member of a group is waiting for another member including itself, to take action (e.g. releasing lock)
- Conditions for Deadlock (All four must be met)
  - **Mutual Exclusion**
    - Occurs when threads claim exclusive control of resources that they require (e.g. thread grabing a lock)
  - **Hold-and-wait**
    - Occurs when threads hold resources allocated to them (e.g locks that they have already acquired) while waiting for additional resources (e.g. locks that they wish to acquire)
  - **No Preemption**
    - Occurs when resource cannot be forcibly removed from threads that are holding them
  - **Circular Wait**
    - Occurs when there exists a circular chain of threads such that each threads hold one or more resources (e.g. locks) that are being requested by the next thread in the chain.

Example



- Preventions
  - **Circular Wait**
    * Write code such that circular wait is never induced
    * Is the most practical prevention technique
    * Requires deep understanding of the code base
    * **Total Ordering** (Most starightforward)

      <u>**Example**</u>

      Given two locks in the system (`L1` and `L2`), always acuiqure `L1` before `L2`

    * **Partial Ordering** (Applied to complex systems)

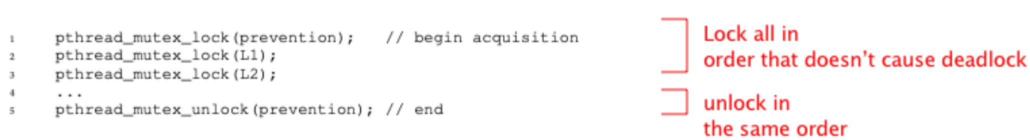
      <u>**Example**</u>

      Memory mapping code in Linux (has then different groups).

      (Simple) `i_mutex` before `i_mmap_mutex`

      (More complex) `i_mmap_mutex` before `private_lock` before `swap_lock` before `mapping->tree_lock`
  - **Hold-and-wait**
    * Can be avoided by acquiring all locks at once
    * Can be problematic
    * Must know which lock must be held and acquire ahead of time
    * Is likely to decrease concurrency (since all need to be acquired over their needs)

      <u>**Example**</u>



- **No Preemption**
  - Can be avoided by adding code that force unlock if not available

    Thread 1

```
top:
  pthread_mutex_lock(L1);
  if (pthread_mutex_trylock(L2) != 0) {
    pthread_mutex_unlock(L1);
    goto top;
  }
```

1. Check if L2 is locked or not available

2. Unlock L1 forcibly (to avoid deadlock)

    Thread 2

```
top:
  pthread_mutex_lock(L2);
  if (pthread_mutex_trylock(L1) != 0) {
    pthread_mutex_unlock(L2);
    goto top;
  }
```

  - `pthread_mutex_trylock` tries to lock the speicied mutex.
  - `pthread_mutex_trylock` returns 0 if lock is available
  - `pthread_mutex_trylock` returns the following error if occupied

    `EBUSY` - Mutex is already locked

    `EINVAL` - Is not initialized mutex

    `EFAULT` - Is in valid pointer
  - May result in **live lock**
- **Mutual Exclusion**
  - Idea: Avoid the mutual exclusion at all
  - Use **lock-free/wait-free** approach: building data structures in a manner that does not require explicit locking using hardware instructions

    Example

```
int CompareAndSwap(int *address, int expected, int new) {
  if (*address == expected) {
    *address = new;
    return 1; // success
  }
  return 0; // failure
}
```

- Avoidance
  - **Banker's Algorithm**

24 Process

- Is a program in execution
- Is named by it's process ID or PID
- Can be described by the following states at any point in time
  - Address Space
  - CPU Registers
  - Program Counter
  - Stack Pointer
  - I/O Information

  (wait. this is PCB)
- Exists in one of many different **process states**, including
  1. Running
  2. Ready to Run
  3. Blocked

  - Different events (Getting Scheduled, descheduled, or waiting for I/O) transitions one of these states to the other

References

1) Coding Horror, Understanding User and Kernel Mode, link

2) Kansas State University, Basics of How Operating Systems Work, link

3) Kansas State University, Glossary, link

4) Tutorials Point, User-level threads and Kernel-level threads, link

5) Science Direct, Scheduling Policy, link

6) Guru 99: What is CPU Scheduling?, link

7) Wikipedia: Round-robin Scheduling, link

8) Kansas State University, The Process Scheduler, link