The University for
world-class professionals

Manchester
Metropolitan
University

Advanced Operating Systems (6G7Z1004)
Lecture: Concurrency

Soufiene Djahel
Office: John Dalton E114

Email: s.djahel@mmu.ac.uk

Telephone: 0161 247 1522

Office hours: Monday 13 -15, Tuesday 11-12

This lecture was adapted from the slides originally designed by Prof. Rene Doursat, MMU, UK

Recap from last week

- OS principles
- Processes, threads and scheduling
- Concurrency
- Memory management
- File system management
- Privacy and Security

a. Processes (Summary)
   - ✓ What is a process?
   - ✓ Process states
   - ✓ Process description
   - ✓ Process control

b. Threads (Summary)

c. Concurrency

d. Deadlock

a. Processes (Summary)

b. What is a process?

- A process is the <u>activity</u> of executing a program

Pasta for six

- boil 1 quart salty water
- stir in the pasta
- cook on medium until "al dente"
- serve

thread of execution

Program

Process

a. Processes (Summary)
What is a process?

- It can be interrupted to let the CPU execute a higher-priority process

Pa

- boil wat
- stir
- coo unt
- serv

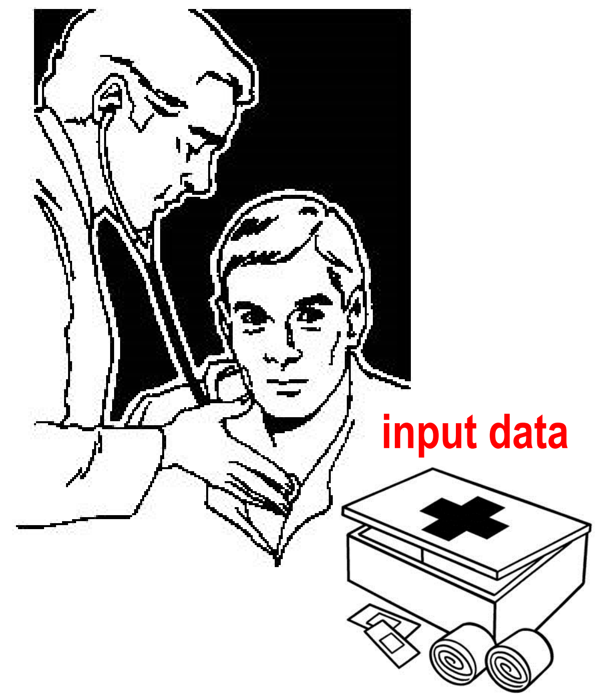
First aid

- – Get the first aid kit
- – Check pulse
- – Clean wound with alcohol
- – Apply band aid

thread of execution

Program

CPU (changes hat to “doctor”)



input data

Process

a. Processes (Summary)

What is a process?

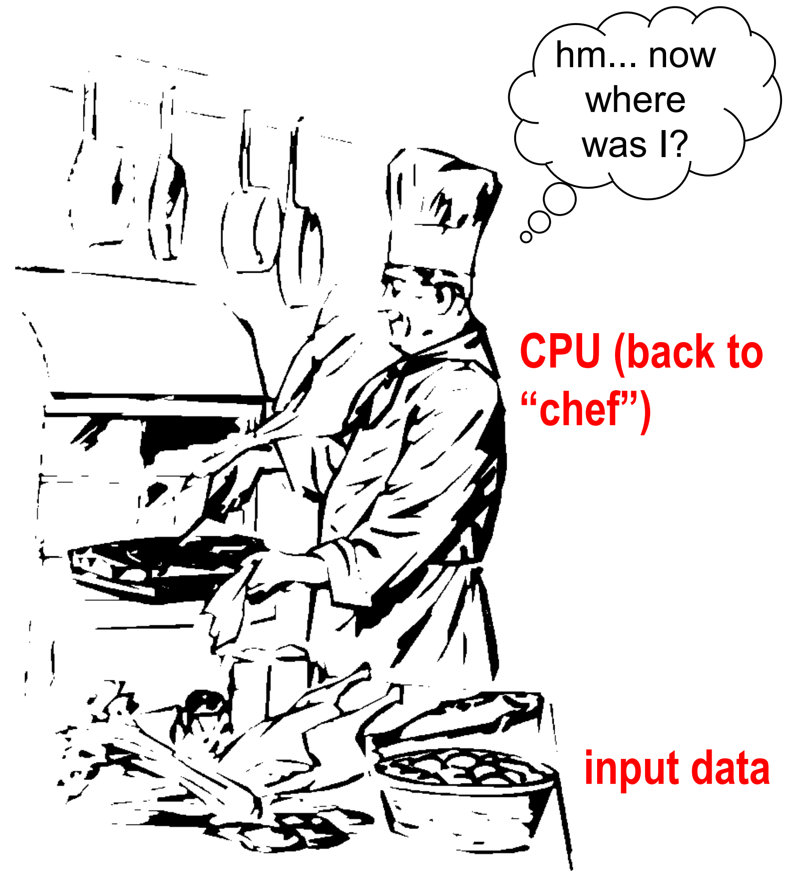
- . . . and then resumed exactly where the CPU left off

Pasta for six

- *boil 1 quart salty water*
- *stir in the pasta*
- *cook on medium until "al dente"*
- *serve*

thread of execution

Program



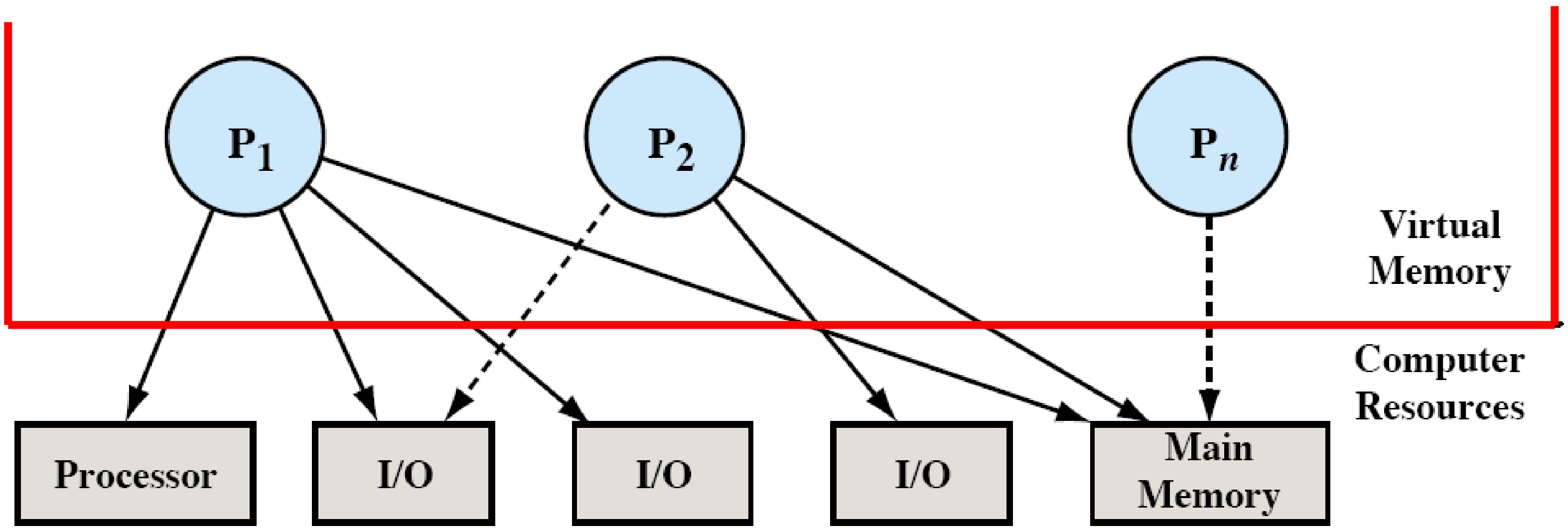
CPU (back to "chef")

input data

Process

a. Processes (Summary)
What is a process?

- The OS has to multiplex resources to the processes
  - ✓ a number of processes have been created
  - ✓ each process during the course of its execution needs access to system resources: CPU, main memory, I/O devices

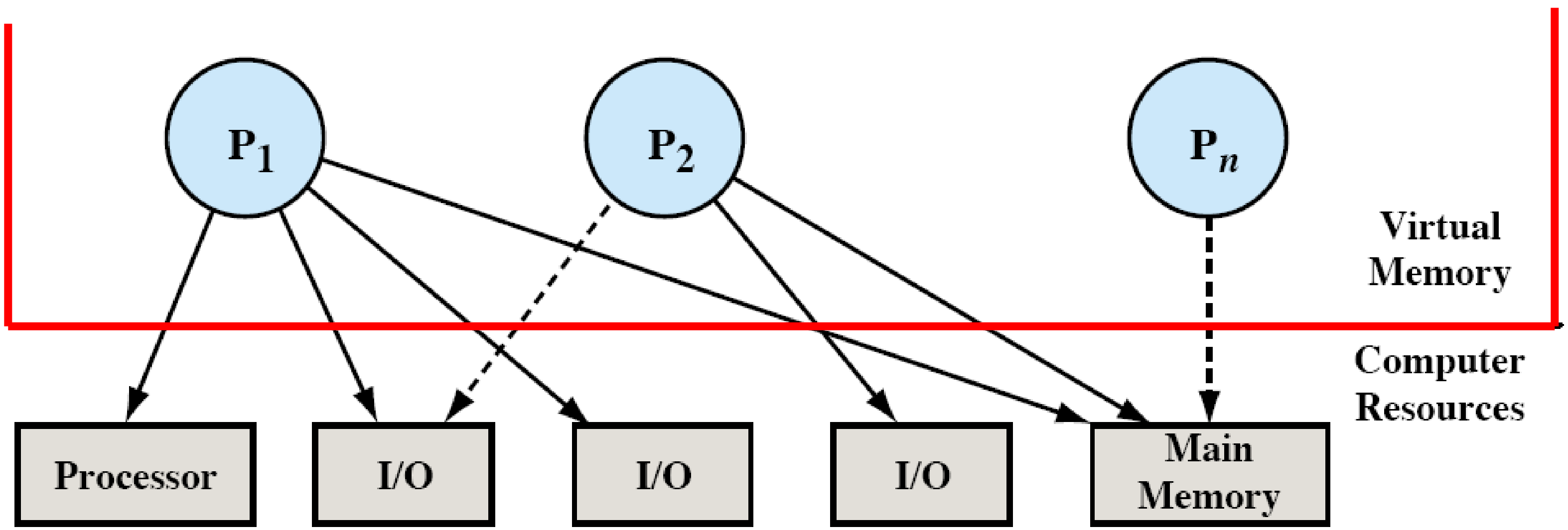


Stallings, W. *Operating Systems: Internals and Design Principles.*

Resource allocation for processes (one snapshot in time)

a. Processes (Summary)
What is a process?

- P_1 is running; at least part of the process is in main memory, and it has control of two I/O devices
- P_2 is also in main memory but is blocked waiting for an I/O device allocated to P_1
- P_n has been swapped out and is therefore suspended

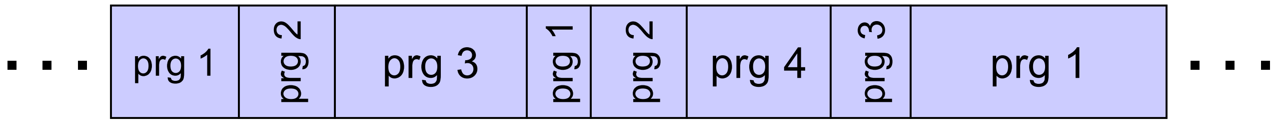


Stallings, W. *Operating Systems: Internals and Design Principles.*

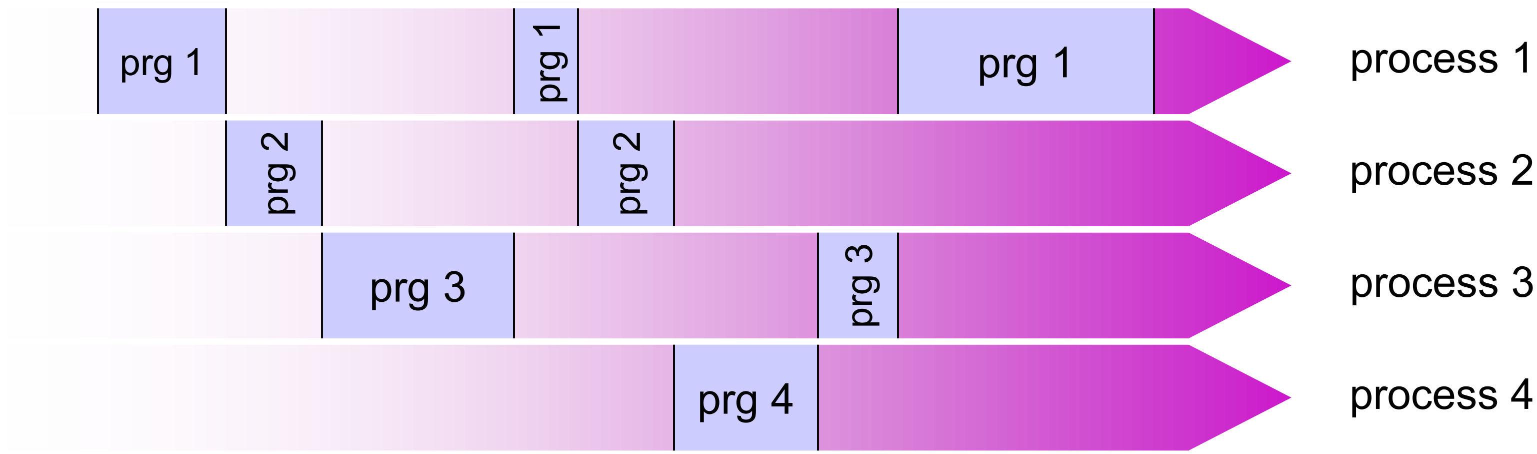
Resource allocation for processes (one snapshot in time)

a. Processes (Summary)
What is a process?

- Multitasking can be conveniently described in terms of multiple processes running in (pseudo) parallel



(a) Multitasking from the CPU's viewpoint



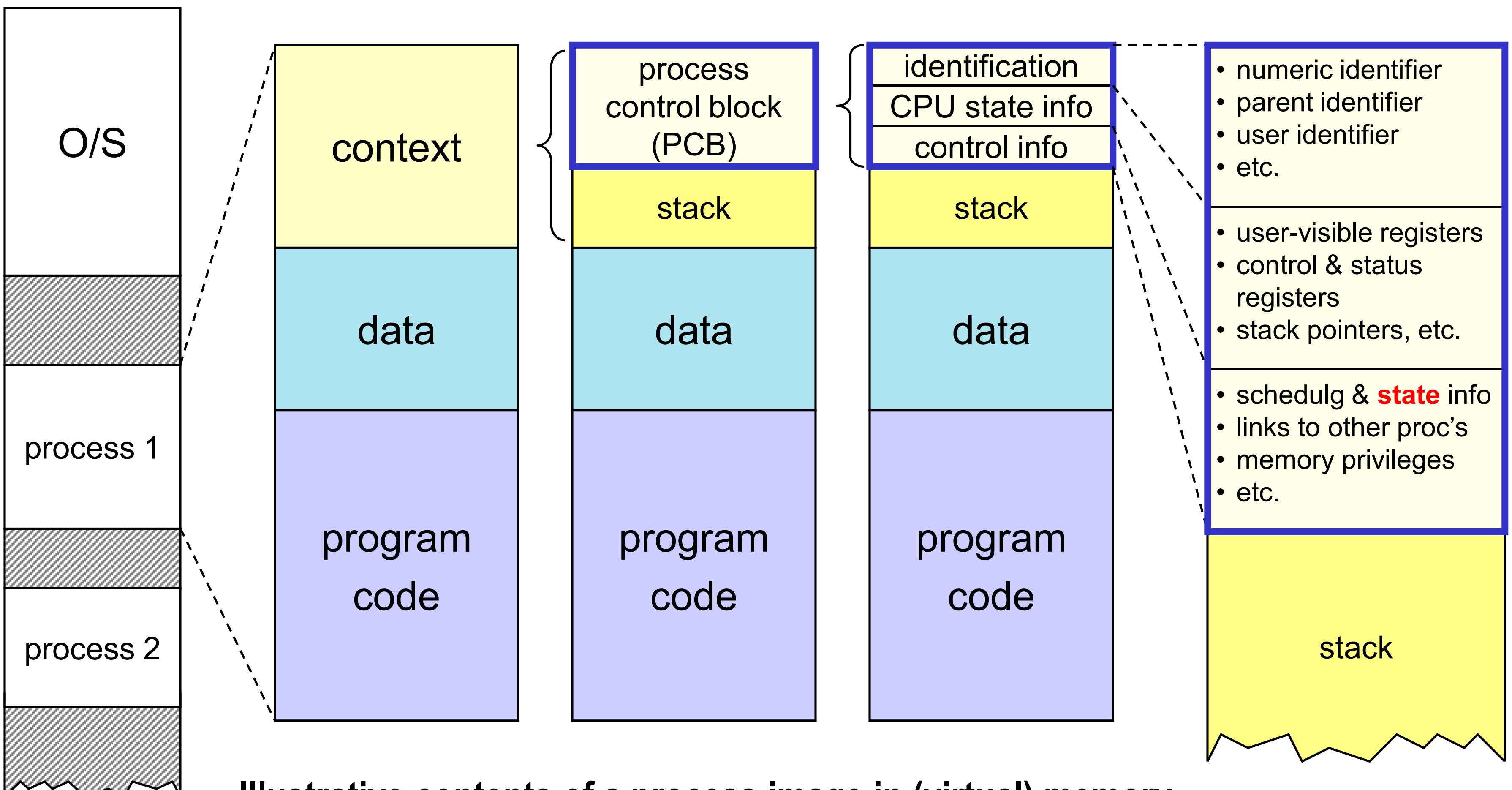
(b) Multitasking from the processes' viewpoint = 4 virtual program counters

Pseudo-parallelism in multitasking

a. Processes (Summary)

Process description

- In the process table, the OS keeps one structure per process, the *Process Control Block* (PCB), containing:



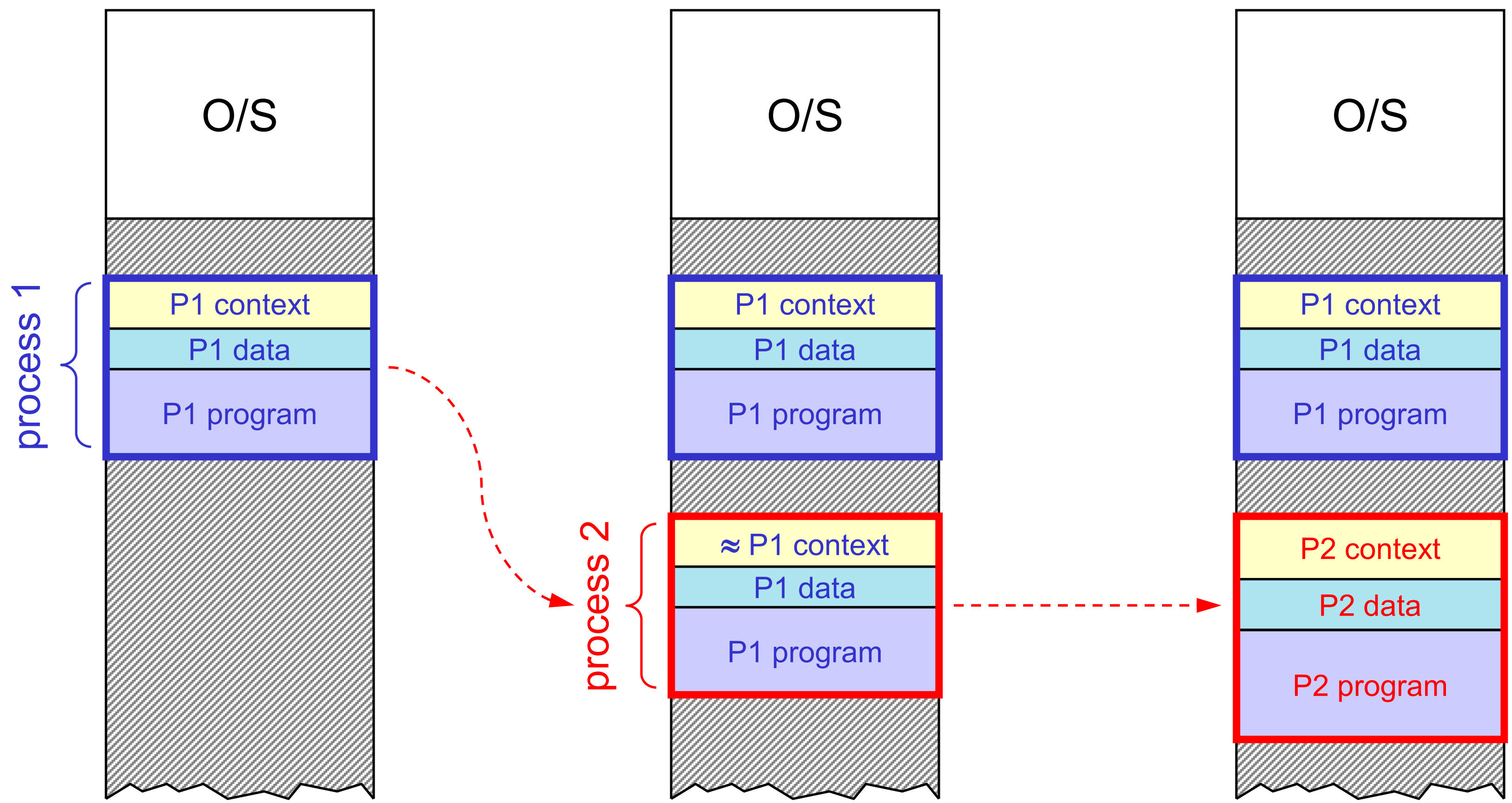
Illustrative contents of a process image in (virtual) memory

a. Processes (Summary)

Process control

1. Clone child process
   - ✓ `pid = fork()`

2. Replace child's image
   - ✓ `execve(name, ...)`



a. Processes (Summary)

Process control

```
...
int main(...)                    // the same program will be executed twice, but differently (not concurrently)
{
    ...
    if ((pid = fork()) == 0)                                    // create a process:
    {                                                           // we are in the child process
        fprintf(stdout, "Child pid: %i\n", getpid());
        err = execvp(command, arguments);                       // execute child process

        fprintf(stderr, "Child error: %i\n", errno);
        exit(err);
    }
    else if (pid > 0)                                           // we are in the parent process
    {                                                           // and pid is the child's ID
        fprintf(stdout, "Parent pid: %i\n", getpid());
        pid2 = waitpid(pid, &status, 0);                        // wait for child process
        ...
    }
    ...

    return 0;
}
```

Implementing a shell command interpreter
by process spawning

a. Processes (Summary)

Process control

- What events trigger the OS to switch processes?
  - ✓ **interrupts** — external, asynchronous events, independent of the currently executed process instructions
    - Clock interrupt → OS checks time and may block process
    - I/O interrupt → data has come, OS may unblock process
    - Memory fault → OS may block process that must wait for a missing page in memory to be swapped in

a. Processes (Summary)

Process control

Traps

✓**exceptions** — internal, <u>involuntary synchronous</u> events caused by instructions

→ OS may terminate or recover process

✓**system calls** — internal, <u>voluntary synchronous</u> events calling a specific OS service

→ after service completed, OS may either resume or block the calling process, depending on I/O, priorities, etc.

a. Processes (Summary)

b. Threads (Summary)
   - ✓ Separation of resource ownership and execution
   - ✓ Same old throughput story
   - ✓ Implementation of threads
   - ✓ Practical uses of multithreading

c. Concurrency

d. Deadlock

Threads

Separation of resource ownership and execution

- In fact, a process embodies two independent concepts
  1. resource ownership
  2. execution & scheduling

1. Resource ownership
   - ✓ a process is allocated address space to hold the image, and is granted control of I/O devices and files
   - ✓ the OS prevents interference among processes while they make use of resources (multiplexing)
2. Execution & scheduling
   - ✓ a process follows an execution path through a program
   - ✓ it has an execution state and is scheduled for dispatching

Threads

Separation of resource ownership and execution

- The execution part is a "thread" that can be multiplied

Pasta for six

- boil 1 quart salty water
- stir in the pasta
- cook on medium until "al dente"
- serve

other thread

thread of execution

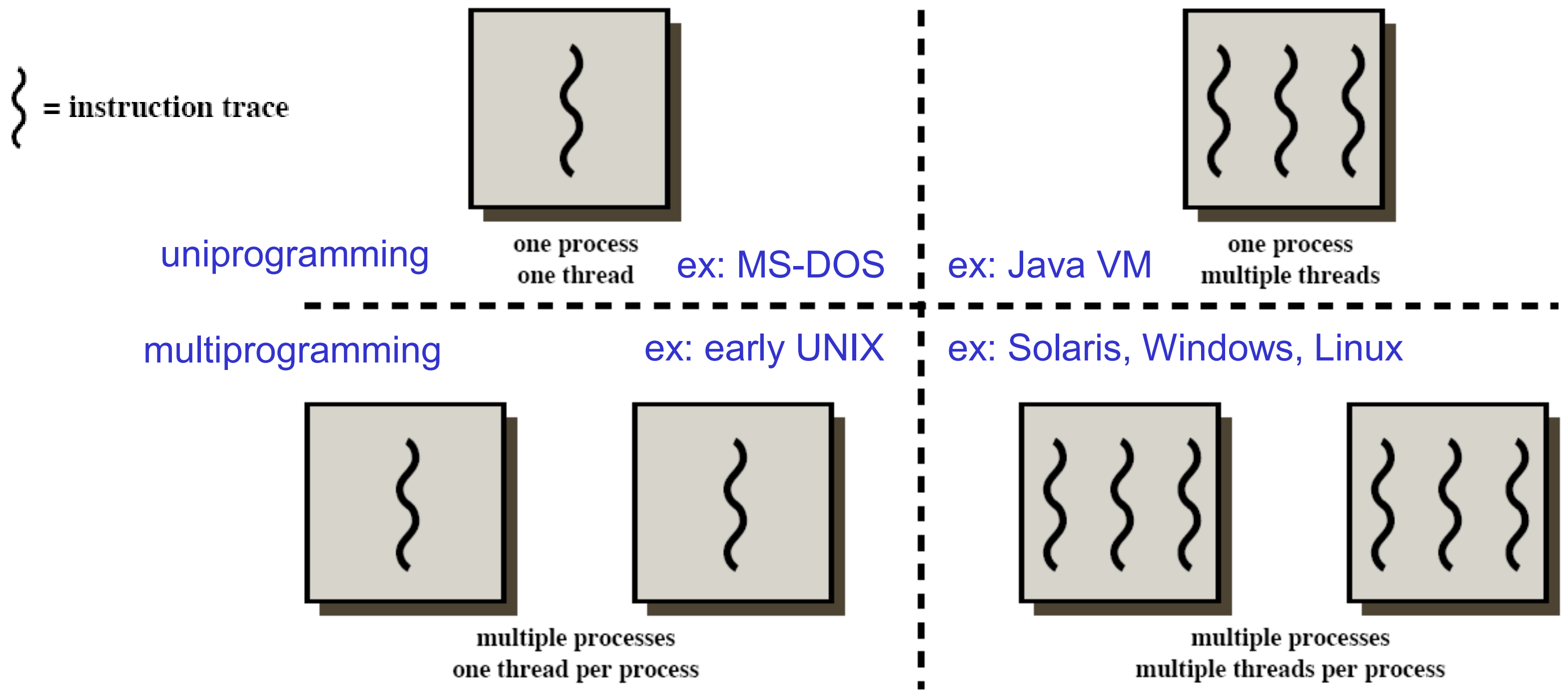
Program

Process

Threads

Separation of resource ownership and execution

- **Multithreading**
  - ✓ refers to the ability of an OS to support multiple threads of execution within a single process



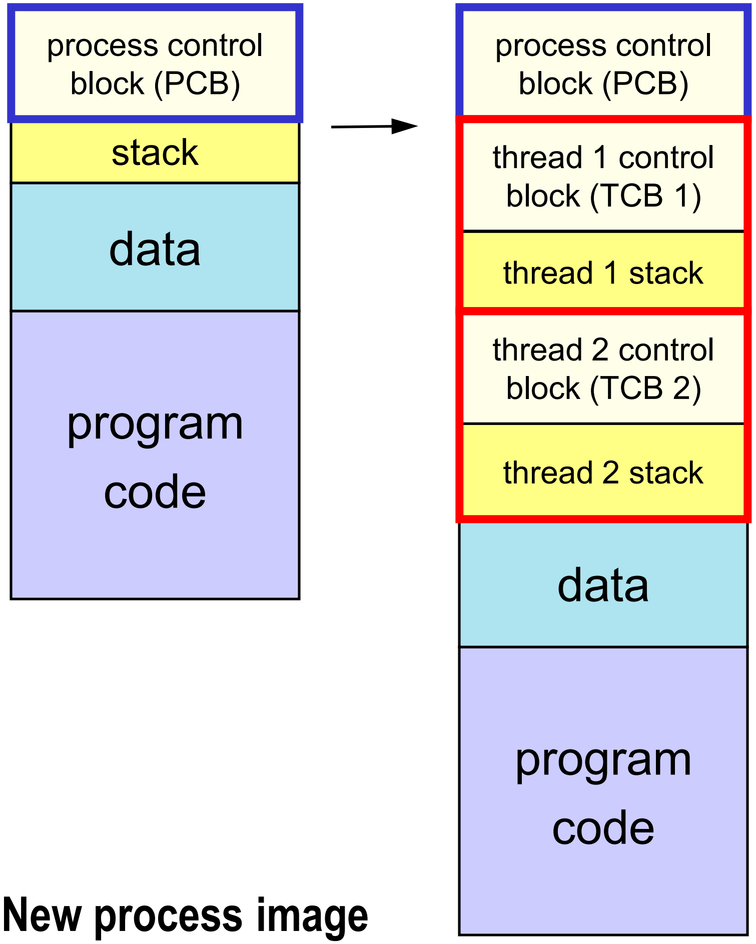
Process-thread relationships

Stallings, W. *Operating Systems: Internals and Design Principles.*

b. Threads (Summary)

Separation of resource ownership and execution

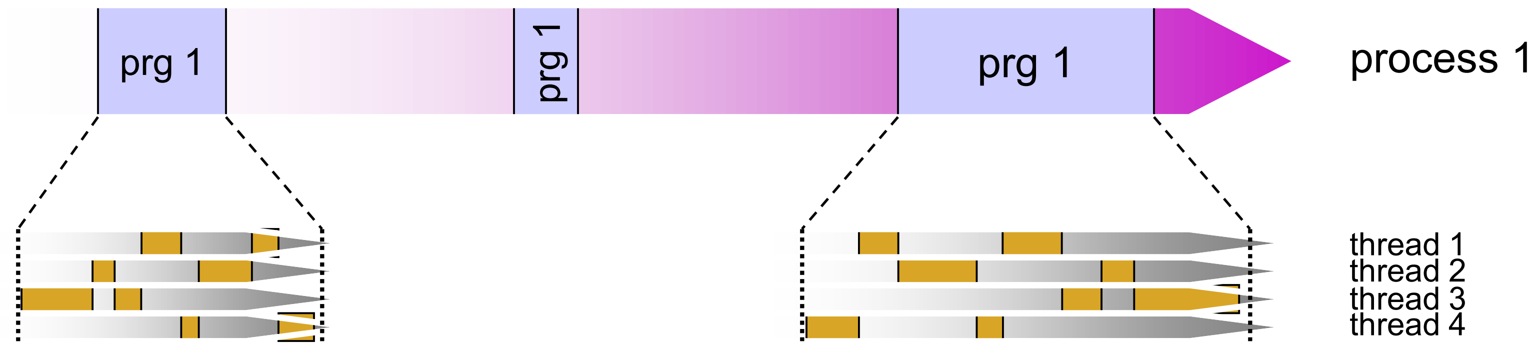
- Multithreading requires changes in the process description model
  - ✓ each thread of execution receives its own control block and stack
    - own execution state ("Running", "Blocked", etc.)
    - own copy of CPU registers
    - own execution history (stack)
  - ✓ the process keeps a global control block listing resources currently used



New process image

b. Threads (Summary)
Same old throughput story

- **The same multitasking idea applies in multithreading**
  - ✓ multithreading is basically the same as multitasking at a finer level of temporal resolution (and within the same address space)
  - ✓ the same illusion of parallelism is achieved at a finer grain

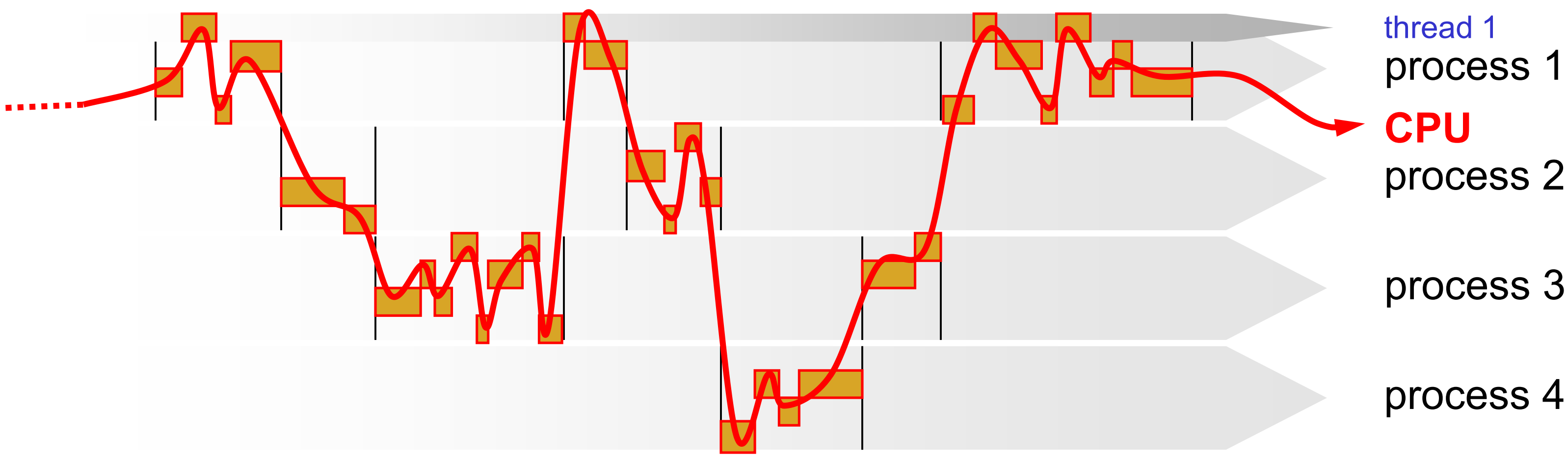


Multithreading

b. Threads (Summary)

Same old throughput story

- **The same multitasking idea applies in multithreading**
  - ✓ in a single-processor system, there is still only one CPU going through all the threads of all the processes



Multithreading

b. Threads (Summary)

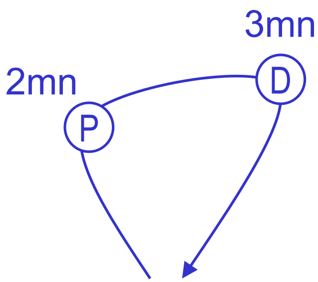
Same old throughput story

- Benefits of multithreading compared to multitasking
  - ✓ it takes less time to create a new thread than a new process
  - ✓ it takes less time to terminate a thread than a process
  - ✓ it takes less time to switch between two threads within the same process than between two processes
  - ✓ threads within the same process share memory and files, therefore they can communicate with each other without having to invoke the kernel
  - ✓ for these reasons, threads are sometimes called "lightweight processes"

  → *if an application should be implemented as a set of related executions, it is far more efficient to use threads than processes*

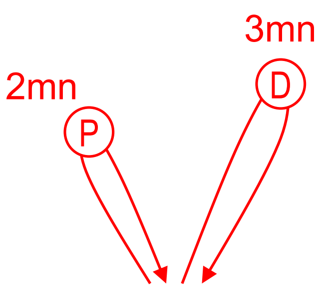
b. Threads (Summary)
Practical uses of multithreading

- **Illustration: two shopping scenarios**
  - Single-threaded shopping



    - you are in the grocery store
    - first you go to produce and grab salad and apples, then you go to dairy and grab milk, butter and cheese
    - it took you about 1mn x 5 items = 5mn

  - Multithreaded shopping



    - you take your two kids with you to the grocery store
    - you send them off in two directions with two missions, one toward produce, one toward dairy
    - you wait for their return for a maximum duration of about 1mn x 3 items = 3mn

b. Threads (Summary)
Practical uses of multithreading

- **Examples of real-world multithreaded applications**
  - ✓ Web client (browser)
    - must download page components (images, styles, etc.) simultaneously; cannot wait for each image in series
  - ✓ Web server
    - must serve pages to hundreds of Web clients simultaneously; cannot process requests one by one
  - ✓ Word processor, spreadsheet
    - provides uninterrupted GUI service to the user while reformatting or saving the document in the background

→ *again, same principles as time-sharing (illusion of interactivity while performing other tasks), this time inside the same process*

a. Processes (Summary)

b. Threads (Summary)

c. Concurrency

   - ✓ Types of process interaction
   - ✓ Race conditions & critical sections
   - ✓ Mutual exclusion by busy waiting
   - ✓ Mutual exclusion & synchronization

d. Deadlock

c. Concurrency

Types of process interaction

- Concurrency refers to any form of interaction among processes or threads
  - ✓ concurrency is a fundamental part of OS design
  - ✓ concurrency includes
    - communication among processes/threads
    - sharing of, and competition for, system resources
    - cooperative processing of shared data
    - synchronization of process/thread activities
    - organized CPU scheduling
    - solving deadlock and starvation problems

c. Concurrency

Types of process interaction

- Concurrency arises in the same way at different levels of execution streams

Software view

- ✓ **multiprogramming** — interaction between multiple processes running on one CPU (pseudo-parallelism)
- ✓ **multithreading** — interaction between multiple threads running in one process

Hardware view

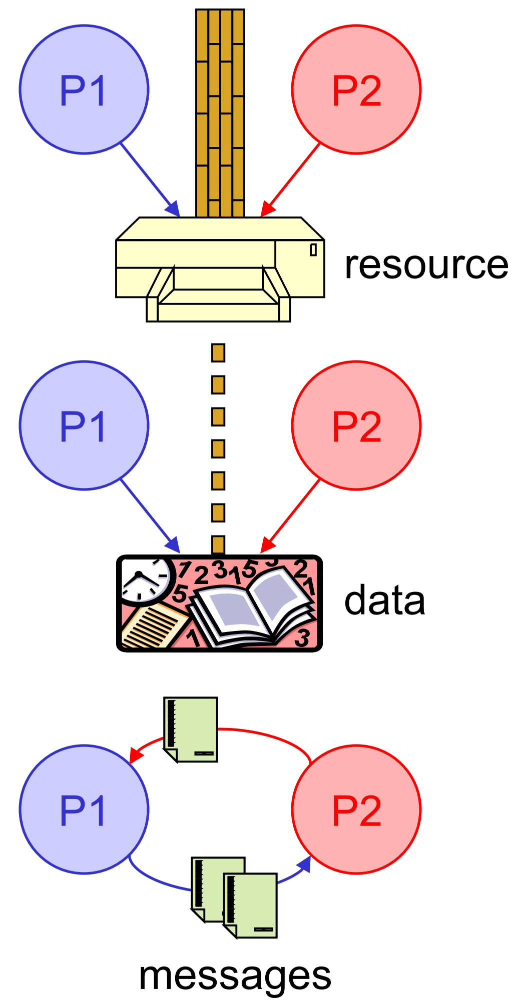
- ✓ **multiprocessors** — interaction between multiple CPUs running multiple processes/threads (real parallelism)
- ✓ **multicomputers** — interaction between multiple computers running distributed processes/threads

→ *the principles of concurrency are basically the same in all of these categories*

c. Concurrency

Types of process interaction

- Whether processes or threads: three basic interactions
  - ✓ **processes unaware of each other** — they must use shared resources independently, without interfering, and leave them intact for the others
  - ✓ **processes indirectly aware of each other** — they work on common data and build some result together via the data
  - ✓ **processes directly aware of each other** — they cooperate by communicating, e.g., exchanging messages



c. Concurrency
Race conditions & critical sections

- **Race condition:** a situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
  - ✓ there is a "race condition" if the outcome depends on the order of the execution
- **Critical section (region):** a section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
- **Mutual exclusion:** the requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

c. Concurrency

Race conditions & critical sections

- **Significant race conditions in I/O & variable sharing**

A

```
char chin, chout;

void echo()
{
  do {
  1 chin = getchar();
  2 chout = chin;
  3 putchar(chout);
  }
  while (...);
}
```

```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

B

```
char chin, chout;

void echo()
{
  do {
  4 chin = getchar();
  5 chout = chin;
  6 putchar(chout);
  }
  while (...);
}
```

```
> ./echo
Hello world!
Hello world!
```

Multithreaded echo (lucky)

lucky
CPU
scheduling

☺

c. Concurrency

Race conditions & critical sections

➢ **Significant race conditions in I/O & variable sharing**

Global variables

```
char chin, chout;

void echo()
{
   do {
 1  chin = getchar();
 5  chout = chin;
 6  putchar(chout);
   }
   while (...);
}
```

A

B

```
char chin, chout;

void echo()
{
   do {
 2  chin = getchar();
 3  chout = chin;
 4  putchar(chout);
   }
   while (...);
}
```

unlucky CPU scheduling

☹

```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

```
> ./echo
Hello world!
ee...
```

Multithreaded echo (unlucky)

c. Concurrency

Race conditions & critical sections

- **Significant race conditions in I/O & variable sharing**

changed to local variables

A

```
void echo()
{
  char chin, chout;

  do {
  1 chin = getchar();
  5 chout = chin;
  6 putchar(chout);
  }
  while (...);
}
```

B

```
void echo()
{
  char chin, chout;

  do {
  2 chin = getchar();
  3 chout = chin;
  4 putchar(chout);
  }
  while (...);
}
```

unlucky CPU scheduling

☹

```
> ./echo
Hello world!
Hello world!
```

Single-threaded echo

```
> ./echo
Hello world!
eH...
```

Multithreaded echo (unlucky)

c. Concurrency

Race conditions & critical sections

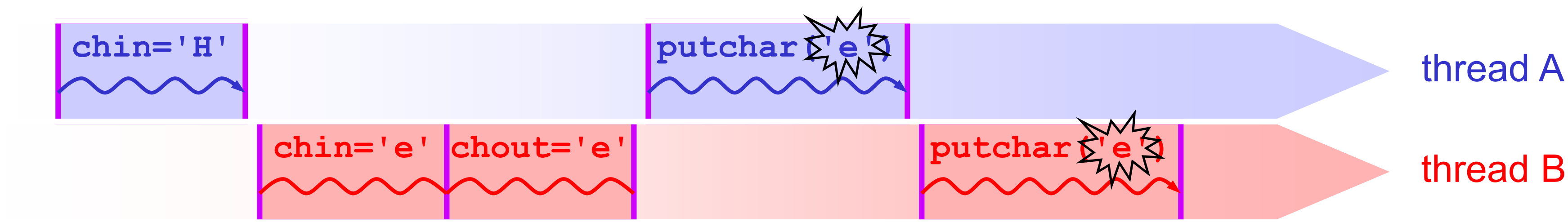
- **Significant race conditions in I/O & variable sharing**
  - ✓ note that, in this case, replacing the global variables with local variables did not solve the problem
  - ✓ we actually had two race conditions here:
    - one race condition in the shared variables and the order of value assignment
    - another race condition in the shared output stream: which thread is going to write to output first (this race persisted even after making the variables local to each thread)

→ generally, problematic race conditions may occur whenever resources and/or data are shared (by processes unaware of each other or processes indirectly aware of each other)

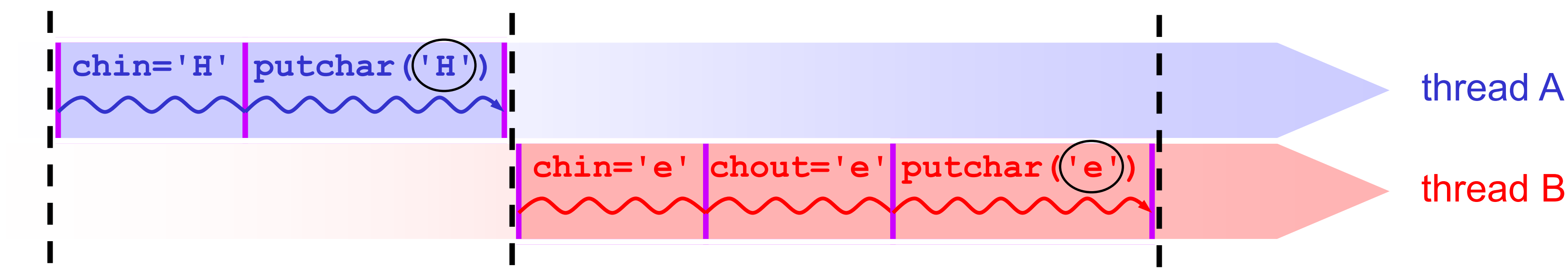
How to avoid race conditions?

c. Concurrency
Race conditions & critical sections

- **How to avoid race conditions?**
  - ✓ find a way to keep the instructions together
  - ✓ this means actually. . . reverting from too much interleaving and going back to “indivisible” blocks of execution!!



(a) too much interleaving may create race conditions



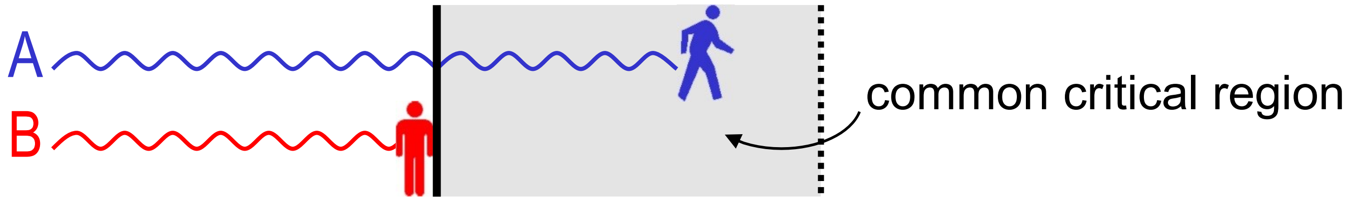
(b) keeping “indivisible” blocks of execution avoids race conditions

c. Concurrency

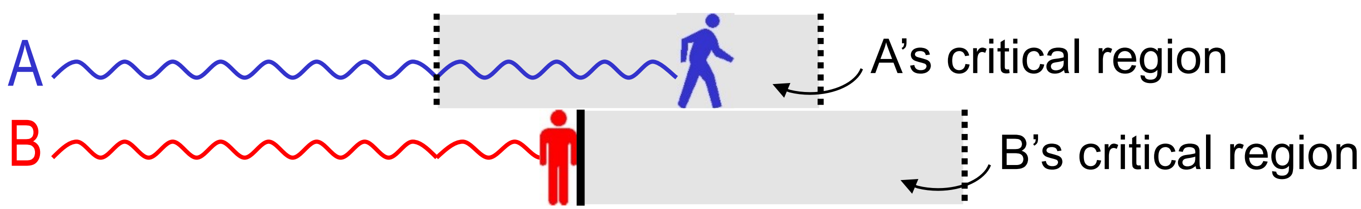
Race conditions & critical sections

- The "indivisible" execution blocks are critical sections

✓ *a critical region is a section of code that may be executed by <u>only one process or thread at a time</u>*



✓ although it does not have to be the same region of memory or section of program in both processes



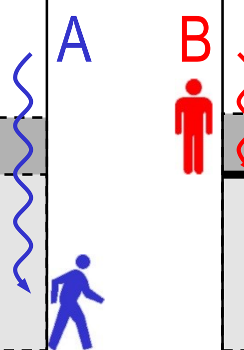
→ *but physically different or not, what matters is that these regions <u>cannot</u> be interleaved or executed in parallel (pseudo or real)*

c. Concurrency

Race conditions & critical regions

- **We need <u>mutual exclusion</u> from critical regions**
  - ✓ critical regions can be protected from concurrent access by padding them with entrance and exit gates:
    - ✓ a thread must try to check in, then it must check out

```
void echo()
{
   char chin, chout;
   do {
      enter critical region?
      chin = getchar();
      chout = chin;
      putchar(chout);
      exit critical region
   }
   while (...);
}
```




```
void echo()
{
   char chin, chout;
   do {
      enter critical region?
      chin = getchar();
      chout = chin;
      putchar(chout);
      exit critical region
   }
   while (...);
}
```

c. Concurrency

Race conditions & critical regions

Chart of mutual exclusion

1. **mutual exclusion inside** — only one process at a time may be allowed in a critical region

2. **no exclusion outside** — a process stalled in a *non*-critical region may *not* exclude other processes from their critical regions

3. **no indefinite occupation** — a critical region may be only occupied for a finite amount of time

c. Concurrency

Race conditions & critical regions

Chart of mutual exclusion (cont'd)

4. **no indefinite delay** — a process may be only excluded for a finite amount of time (no deadlock or starvation)

5. **no delay when about to enter** — a critical region free of access may be entered immediately by a process

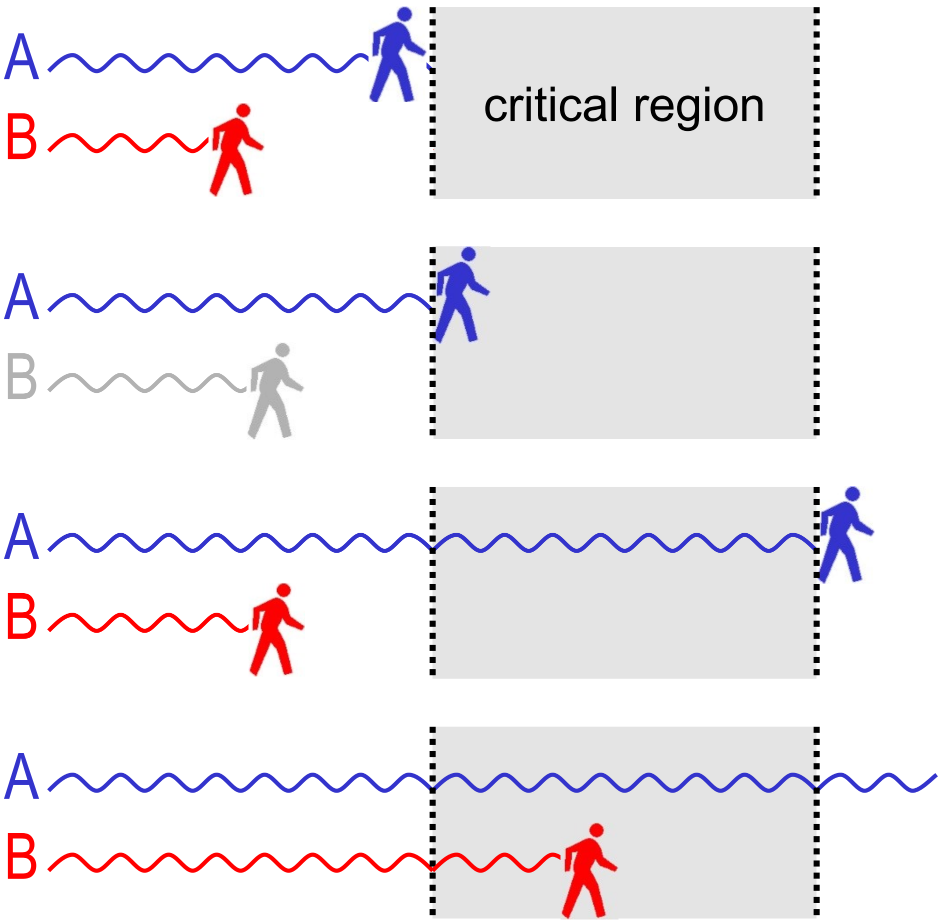
6. **nondeterministic scheduling** — no assumption should be made about the relative speeds of processes

c. Concurrency

Mutual exclusion by busy waiting

- **Disabling hardware interrupts**

1. thread A reaches the gate to the critical region (CR) before B
2. as soon as A enters CR, it disables all interrupts, thus B cannot be scheduled
3. as soon as A exits CR, it reenables interrupts; B can be scheduled again
4. thread B enters CR



c. Concurrency

Mutual exclusion by busy waiting

➢ Disabling hardware interrupts

- ✓ it works, but is foolish
- ✓ what guarantees that the user process is going to ever exit the critical region?
- ✓ meanwhile, the CPU cannot interleave any other task, even unrelated to this race condition
- ✓ the critical region becomes one *physically* indivisible block, not logically.
- ✓ also, this is not working in multi-processors

```
void echo()
{
   char chin, chout;
   do {
      disable hardware interrupts
      chin = getchar();
      chout = chin;
      putchar(chout);
      reenable hardware interrupts
   }
   while (...);
}
```

c. Concurrency

Mutual exclusion by busy waiting

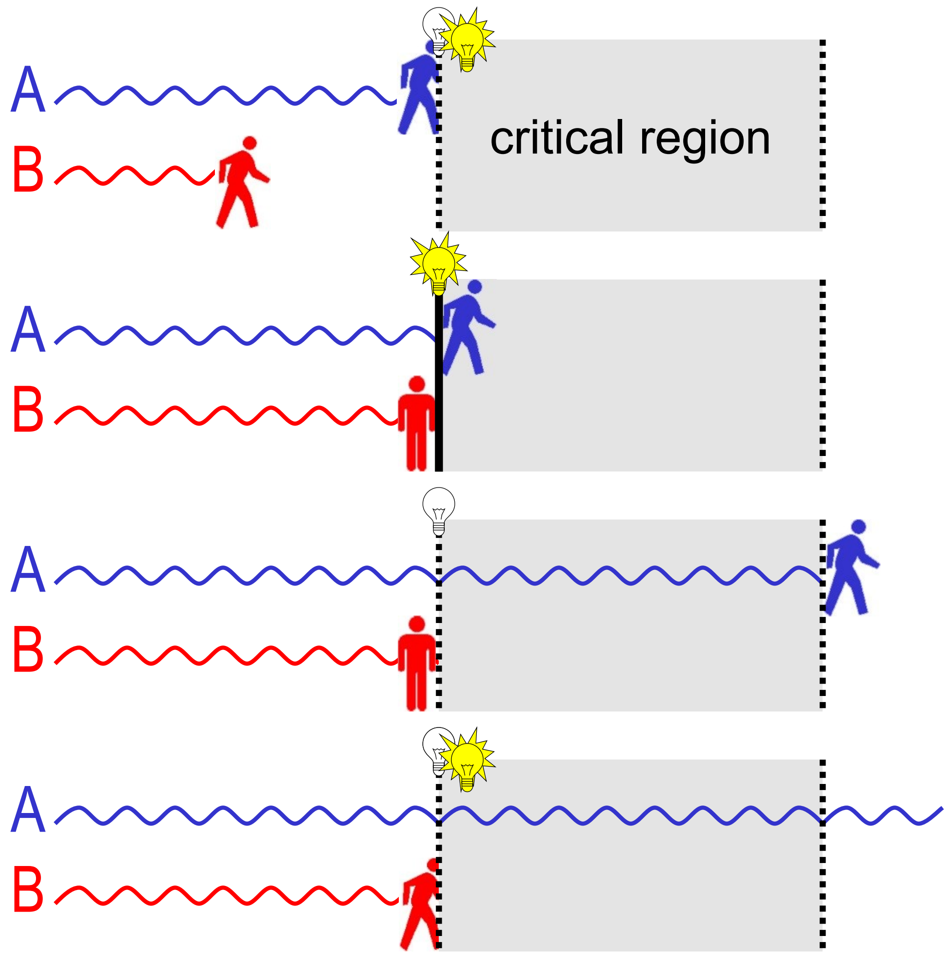
- **Indivisible lock variable**

1. thread A reaches CR and <u>finds the lock at 0 *and* sets it in **one shot**</u>, then enters

1.1' even if B comes right behind A, it will find that the lock is already at 1

2. thread A exits CR, then <u>resets lock to 0</u>

3. thread B finds the lock at 0 and sets it to 1 in **one shot**, just before entering CR



c. Concurrency

Mutual exclusion by busy waiting

- **Indivisible lock variable** 👍
  - ✓ the indivisibility of the "test-lock-and-set-lock" operation can be implemented with a <u>*single*</u> hardware instruction **TSL**

```
enter_region:
    TSL REGISTER,LOCK     | copy lock to register and set lock to 1 (*)
    CMP REGISTER,#0       | was lock zero?
    JNE enter_region      | if it was non zero, lock was set, so loop
    RET                   | return to caller; critical region entered
```

```
leave_region:
    MOVE LOCK,#0          | store a 0 in lock
    RET                   | return to caller
```

```
void echo()
{
   char chin, chout;
   do {
      test-and-set-lock
      chin = getchar();
      chout = chin;
      putchar(chout);
      set lock off
   }
   while (...);
}
```

Tanenbaum, A. S.
Modern Operating Systems.

(*) if lock was already 1, the thread changed nothing and must try again;
if lock was 0, the thread was entitled to change it to 1 and can enter

c. Concurrency

Mutual exclusion by busy waiting

- **Summary of mutual exclusion implementations**
  - **Disabling hardware interrupts**
    - NO: race condition avoided, but can crash the system!
  - **Indivisible lock variable (TSL)**
    - YES: works, but requires special hardware instruction
      - **It is the basis for mutexes**
  - **Other implementations:**
    - **Simple lock variable (unprotected)**
      - NO: still suffers from race condition
    - **Peterson's no-TSL, no-alternation**
      - YES: works in pure software, but processing overhead

c. Concurrency

Mutual exclusion by busy waiting

- **Problem: all implementations rely on <u>busy waiting</u>**
  - ✓ "busy waiting" means that any process/thread (A, B, ...) must continuously execute a tight loop until some condition changes
  - ✓ busy waiting is bad:
    - **waste of CPU time** — the busy process is not doing anything useful, yet remains "Ready" instead of "Blocked"
    - **paradox of inversed priority** — by looping indefinitely, a higher-priority process B may starve a lower-priority process A, thus preventing A from exiting CR and . . . keeping B in the wait! (the CPU is too busy executing an empty loop, i.e. B is working against its own interest)

→ *we need for the waiting process to truly <u>block</u>, not keep idling*

✓Mutual exclusion & synchronization

- Mutexes
- Semaphores
- Monitors

c. Concurrency

Mutual exclusion & synchronization — mutexes

- **Indivisible blocking lock = mutex**
  - ✓ a mutex is a safe lock variable with blocking, instead of tight looping
  - ✓ if **TSL** returns 1, then voluntarily yield the CPU to another thread

```
mutex_lock:
    TSL REGISTER,MUTEX     | copy mutex to register and set mutex to 1
    CMP REGISTER,#0        | was mutex zero?
    JZE ok                 | if it was zero, mutex was unlocked, so return
    CALL thread_yield      | mutex is busy; schedule another thread
    JMP mutex_lock         | try again later
ok: RET                    | return to caller; critical region entered
```

```
mutex_unlock:
    MOVE MUTEX,#0          | store a 0 in mutex
    RET                    | return to caller
```

```
void echo()
{
   char chin, chout;
   do {
      test-and-set-lock or BLOCK
      chin = getchar();
      chout = chin;
      putchar(chout);
      set lock off
   }
   while (...);
}
```

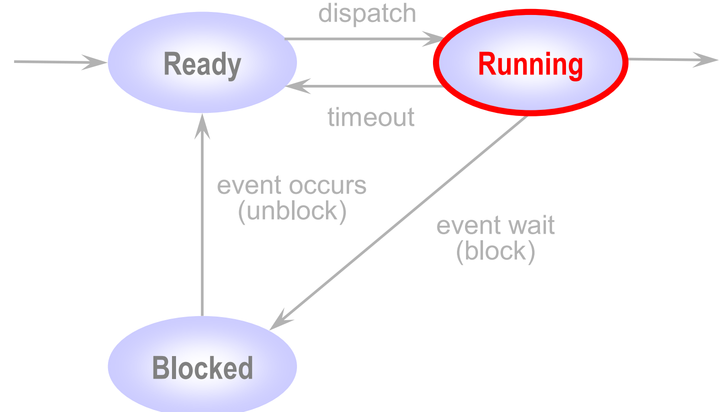
~~JNE enter_region~~ ~~| if it was non zero, lock was set, so loop~~ [redacted]

c. Concurrency

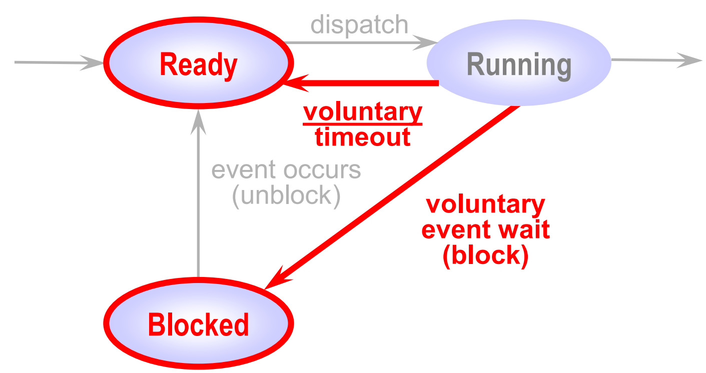
Mutual exclusion & synchronization — mutexes

➢ Difference between <u>busy waiting</u> and <u>blocked</u>

- ✓ in <u>busy waiting</u>, the PC (Program Counter) is always looping (increment & jump back)
- ✓ it can be preemptively interrupted but will loop again tightly whenever rescheduled → *tight polling*



- ✓ when <u>blocked</u>, the process's PC stalls after executing a "yield" call
- ✓ either the process is only timed out, thus it is "Ready" to loop-and-yield again → *sparse polling*
- ✓ or it is truly "Blocked" and put in event queue → *condition waiting*



c. Concurrency

Mutual exclusion & synchronization — Semaphores

- Special integer variable with only three possible operations:
  - ✓ **Initialisation** (to a non-negative number (0 or 1 for binary sem))
  - ✓ **wait**, which decrements the value
    - ✓ If the value is then **negative**, the process calling wait becomes **blocked**
  - ✓ **signal**, which increments the value
    - ✓ If the value is then **<= 0**, a process which has previously been blocked by this semaphore becomes unblocked
  - ✓ Important: wait and signal should be implemented as **atomic (indivisible) primitives**

c. Concurrency

Mutual exclusion & synchronization — Semaphores

- Binary semaphore implementation

```
struct binary_semaphore {
        enum {zero, one} value;
        queueType queue;
};
void semWaitB(binary_semaphore s)
{
        if (s.value == one)
             s.value = zero;
        else {
                   /* place this process in s.queue */;
                   /* block this process */;
        }
}
void semSignalB(semaphore s)
{
        if (s.queue is empty())
             s.value = one;
        else {
                   /* remove a process P from s.queue */;
                   /* place process P on ready list */;
        }
}
```

c. Concurrency

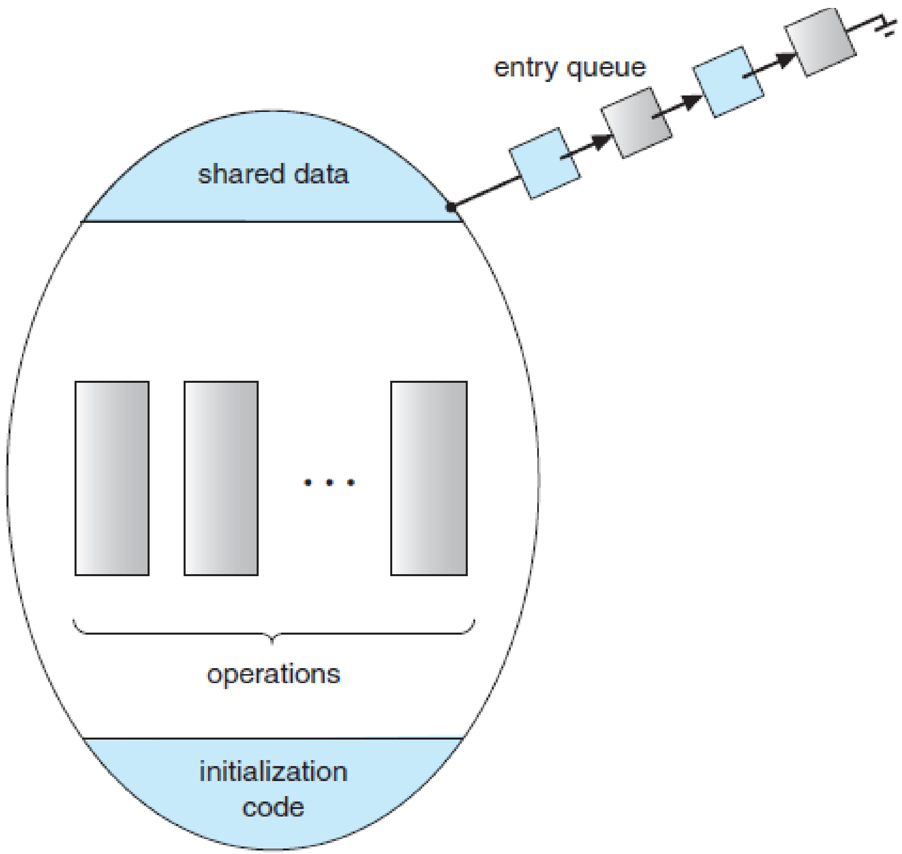
Mutual exclusion & synchronization — Monitors

- Programming language construct that provides equivalent functionality to that of semaphores and is easier to control
- Implemented in a number of programming languages
  - including Concurrent Pascal, Pascal-Plus, and Java
- Monitors encapsulate data structures that are not externally accessible
- Mutually exclusive access to data structure enforced by only "admitting" one process at a time
  - Processes wanting access join a waiting queue

c. Concurrency

Mutual exclusion & synchronization — Monitors

- Schematic view of a monitor



a. Processes (Summary)

b. Threads (Summary)

c. Concurrency

- ✓ Types of process interaction
- ✓ Race conditions & critical sections
- ✓ Mutual exclusion by busy waiting
- ✓ Mutual exclusion & synchronization

d. Deadlock

d. Deadlock
Conditions

- A deadlock situation can arise if the following **4 conditions hold simultaneously**:

1. Mutual exclusion
2. Hold and wait
   - a process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
3. No preemption
   - Resources cannot be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task
4. Circular wait (Potential consequence of the first three)

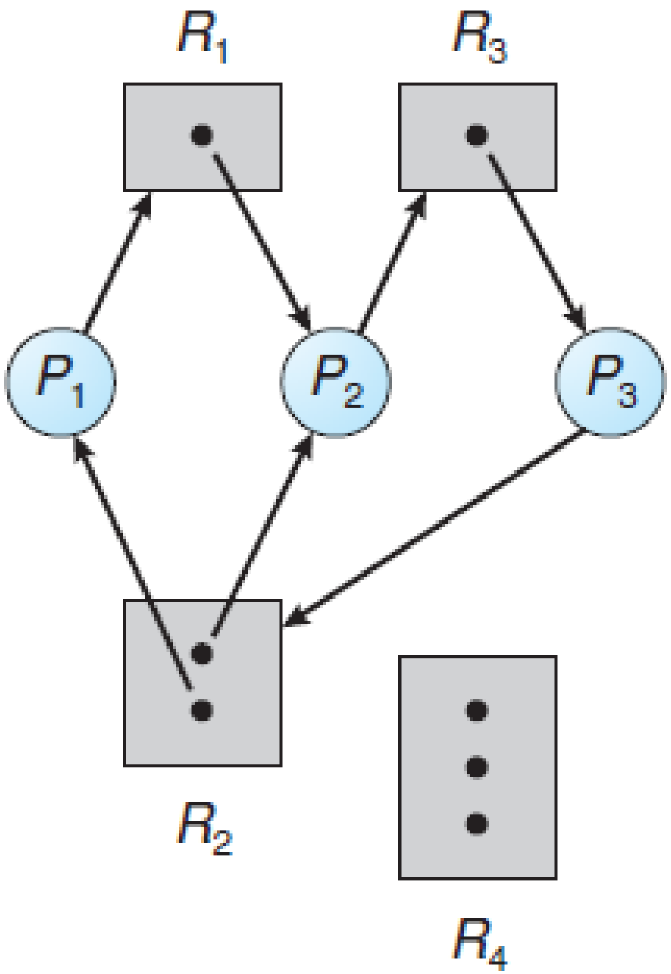
d. Deadlock

Resource Allocation Graph (RAG)

P: process

R: resource

- Instance of a resource



Circular wait: P_1 is waiting for R_1 held by P_2 , P_2 is waiting for R_3 held by P_3, and P_3 is waiting for either P_2 or P_1 to release R_2

d. Deadlock
Example

- Thread A:

  lock X

  lock Y

  perform action involving X and Y

  release X

  release Y

- Thread B:

  lock Y

  lock X

  perform action involving X and Y

  release X

  release Y

- Possible execution:

  A: lock X

  B: lock Y

  A: tries to lock Y

  B: tries to lock X

  DEADLOCK

d. Deadlock
Possible solutions

- Prevent Deadlock
  - adopt a policy that will ensure a deadlock never arises
- Avoid Deadlock
  - avoid entering a state that could lead to deadlock
- Detect Deadlock and Recover

d. Deadlock
Prevention approaches

- Design a system in such a way that the possibility of deadlock is excluded
- Direct or indirect approaches

d. Deadlock

Indirect Approach to Deadlock Prevention

- Cannot prevent mutual exclusion
- Hold and wait: all resources required to complete must be requested at once; if any one is not available, release all locks
- No pre-emption: if a process holding a resource that is required by another process, OS may pre-empt that resource
- Circular wait: define a linear ordering of resources

d. Deadlock

Direct Approach to Deadlock Prevention

- Check that any resource request does not cause a circular wait

d. Deadlock
Deadlock Avoidance

- Process initiation denial
  - Do not start a process if its demands might lead to deadlock.
- Resource allocation denial
  - Do not grant an incremental resource request to a process if this allocation might lead to deadlock
  - Example: banker's algorithm

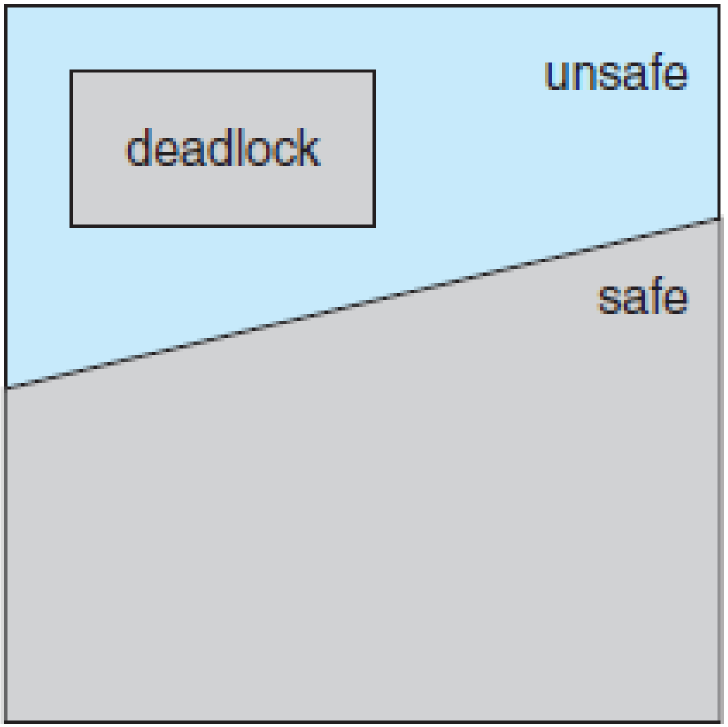
d. Deadlock

Deadlock Avoidance: Banker's Algorithm

- State of the system reflects the current allocation of resources to processes

- Safe state is one in which there is at least one sequence of resource allocations to processes that does not result in a deadlock

- Unsafe state is a state that is not safe



d. Deadlock

Pros and Cons of Deadlock Avoidance

- It is not necessary to preempt and rollback processes, as in deadlock detection
- It is less restrictive than deadlock prevention

- Max. resource requirement for each process must be stated in advance
- Processes must be independent; no synch. requirements
- There must be a fixed number of resources to allocate
- No process may exit while holding resources

d. Deadlock

Deadlock Detection

- Deadlock detection strategies do not limit resource access or restrict process actions
- A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur
- The algorithm is relatively simple (Banker's algorithm)
- Frequent checks consume considerable processor time

d. Deadlock

Deadlock Recovery: possible strategies

- Abort all deadlocked processes
- Back up each deadlocked process to some previously defined checkpoint and restart all processes
- Successively abort deadlocked processes until deadlock no longer exists
- Successively preempt resources until deadlock no longer exists

Next lecture

- Cloud computing